# Logic Regression on High Dimensional Boolean Space

**Poyen Chen**

## 1. Introduction

This problem is one of the problems of 2019 ICCAD contest.

Given a black-boxed 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.

The contestants can feed any input pattern into the generator and get the corresponding output. By viewing the patterns, the contestants should construct a circuit that match the function of the generator as good as possible, while keeping the circuit size (the number of gates in the circuit) as small as possible. The constructed circuit should be represented as a Verilog file.

For detailed description of the problem, please refer to http://iccad-contest.org/2019/Problem_A/2019ICCAD_ProblemA_V5_total.pdf.

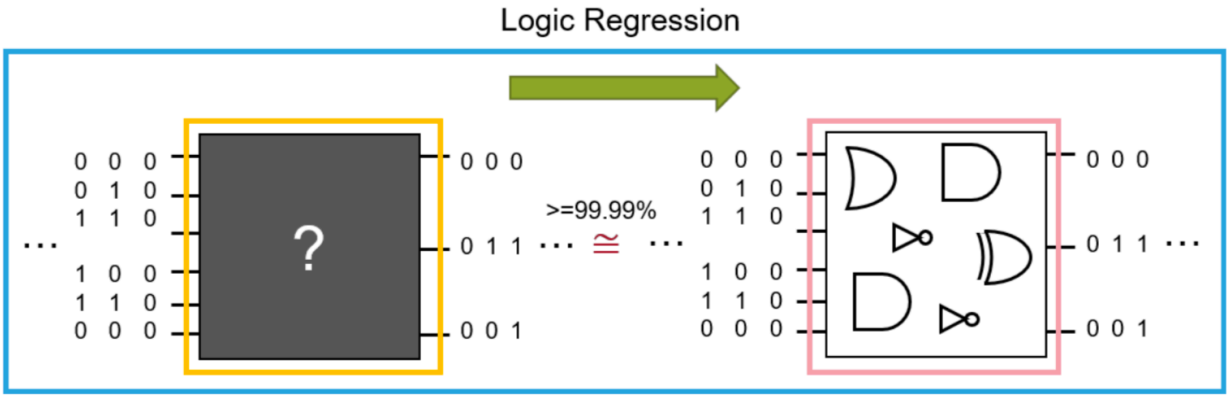



Figure 1. Problem abstract.

## 2. Algorithm Overview

The flow chart of our method is shown in figure 2. In the following passage, we will discuss each step in detail. The algorithm have some adjustable parameters. Changing them will affect the outcome significantly. To investigate the affect of these parameters, some experiments will be provided. For the ease of discussion, the parameters will be in capital letters and will be underlined in the following passage. Like THIS.

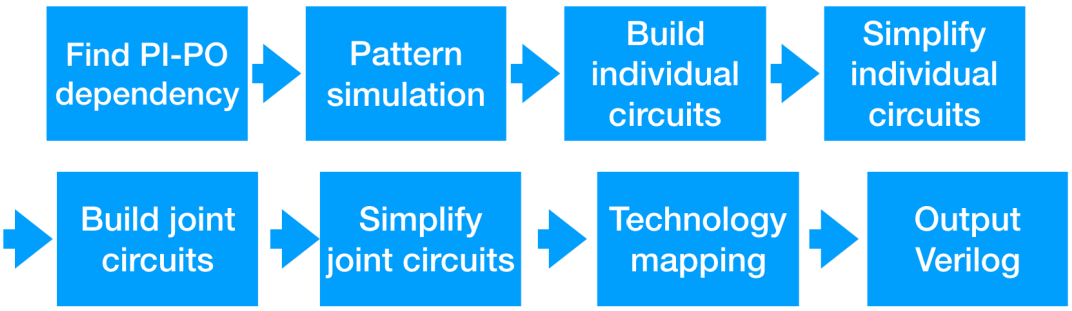



**Figure 2. Flow chart**

(1) Find PI-PO dependency

In the contest, each problem, the number of PIs (primary inputs) ranges from 25 to thousands, while the number of POs (primary outputs) ranges from 1 to 5000. Since it is impossible to feed all possible input patterns into the generator under limited time (3600 seconds each problem in the contest), we have to find smarter ways.

In practice, in a logic circuit, each PO usually does not depend on all PIs. Take the circuit in Figure 3 as an example, X only depends on A and B while Y depends on B and C.

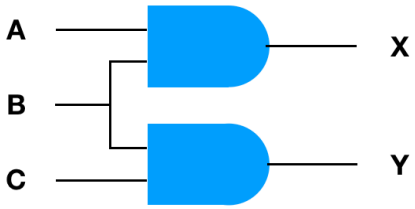



**Figure 3.**

Also, even if a PO depends on all PIs, the dependency of the PO on each PI may not be the same. Take X=A+B*C*D as an example, the dependency of X on A is stronger than its dependency on B, C, and D. That is, flipping A have a higher probability leading to flipping X.

The goal of this step is to find the dependency of each PO on each PI and rank the PI by dependency form the strongest to weakest so that we can reduce the searching space.

For each PI, we random generate a lot of input patterns for other PIs and set the PI as 1 and 0 to observe the pattern of the outputs. Take figure 4 as an example, we are testing the dependency of PO_1 and PO_2 on PI_1, two random pattern [10] and [00] of PI_2 and PI_3 are generated. Since when P1_1 flipped, PO_1 flipped once and PO_2 does not, the dependency of PO_1 on PI_1 is 0.5 and the dependency of PO_2 on PI_1 is 0.

| PI_1 | PI_2 | PI_3 | PO_1 | PO_2 |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |

PI under test | random pattern | dependency 1/2 | dependency 0/2

**Figure 4.**

After this process, we maintain a list of PI dependency for each PO. The list contains all PIs and is sorted from the highest dependency to the lowest.

(2) Pattern simulation

In this step, we feed input patterns to the generator, view the corresponding output patterns, and construct the circuit accordingly. Due to timing constraint, we only enumerate all possible inputs with the top MAXIMUM_SIMULATED_PI dependent PIs, where MAXIMUM_SIMULATED_PI is an adjustable parameter of the algorithm. For other PIs, we random generate the input patterns and treat them as don't-care terms when constructing the circuit.

Take Figure 5 as an example, PI_1, PI_2, PI_3 are the PIs of the circuit. There can be many POs in the circuit but we focus on finding the boolean function of a PO_1 at this time instance. Let MAXIMUM_SIMULATED_PI be 2 and the dependency of PO_1 on PI_1, PI_2, PI_3 be 0.1, 0.5, 0.6 respectively. Since MAXIMUM_SIMULATED_PI is 2, we enumerate all possible input combinations of PI_2 and PI_3, that is, [00], [01], [10], and [11]. The input of PI_1 is randomly generated. After that, we feed the input patterns into the generator and get the corresponding output on PO_1. Then, we treat PI_1 as don't-care PI for PO_1 and build the truth table on the right. Notice that this truth table is not entirely correct for the circuit since PO_1 still have 0.1 dependency on PI_1

so PI_1 cannot be treated as don't-care term. However, when the number of PIs is large, this approximation is necessary with output pattern accuracy as the trade-off.

| PI_1 | PI_2 | PI_3 | PO_1 |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| PI_1 | PI_2 | PI_3 | PO_1 |
|---|---|---|---|
| - | 0 | 0 | 1 |
| - | 0 | 1 | 1 |
| - | 1 | 0 | 1 |
| - | 1 | 1 | 0 |

**Figure 5.**

(3) Build individual circuit

After we constructed the approximated truth table in step (2), we parse it into *pla logic circuit file format* [1] and use *ABC sequential synthesis and verification system* [2] to build the circuit. Notice that there can be multiple POs in the circuit under the black-box generator. However, we build the truth table for each PI individually by step (2) and read them individually to the ABC system. Therefore, we would have several single-PI circuits. All the circuits would be joined together in step (5).

(4) Simplify individual circuit

After building the single-PI circuits in ABC system, we simplify the circuit using a combination of ABC instructions. This will result in approximately 15% less logic gates in the circuit, depending on the amount of time we spend.

(5) Build joint circuit

After simplifying each single-PI circuit, we load all of them into the ABC system to build a multi-PO circuit using the "append" command.

(6) Simplify joint circuit

After building the joint circuit, we perform further circuit simplification. Since the single-PI circuits may have functionally equivalent internal gates with each other, this step can further reduce the circuit size by about 25%. However, this step is removed in the final version of our work. The reason is described in part (3) of the Experiment and Observation section.

(7) Technology mapping

After simplifying the circuit, we map the circuit gates with standard gates. AND, OR, XOR, NAND, NOR, XNOR, buffers, and inverters are used in the mapping.

(8) Output to Verilog

Using the write_verilog command provided by ABC, we can output the mapped circuit into gate level Verilog format.

# 3. Experiments and Observations

The contest organizer provided 7 sample generators for contestants to develop their algorithms. The following discussion will base on these samples. To have better insights, some basic informations of these samples are provided in table 6.

The "Number of dependent PIs for each PO" column denotes how many PIs have dependency strictly greater than zero for each PO. Take the sample named "sample" as an example, the sample have 2 POs. 6 PIs have positive dependency no the first PO and 4 have positive dependency on the second. Thus, "6 4" is recorded in the "Number of dependent PIs for each PO" column. Notice that the number of dependent PIs are determined by random simulation discussed in step (1) in the Algorithm Overview section. Since most cases have a large number of PIs, we can never be 100% sure about how many PIs are dependent for each PO unless we enumerate all possible input patterns. Therefore, the values in the "Number of dependent PIs for each PO" column are estimated. In fact, the value differs slightly each time we run the algorithm. However, this column still gives us an idea of how the POs depend on the PIs since it is the lower

bound of the true number of dependent PIs. The circuit having larger number of dependent PIs in average are generally harder to simulate and usually result in lower accuracy.

Table 6.

| Samle name | Number of PIs | Number of POs | Number of dependent PIs for each PO |
|---|---|---|---|
| **“sample”** | 6 | 2 | 6 4 |
| **case1** | 53 | 19 | 21 24 25 25 25 24 25 24 23 21 20 18 15 12 9 6 3 1 0 |
| **case2** | 121 | 38 | 2 1 2 2 0 2 10 1 6 5 6 10 5 3 6 9 4 6 11 10 6 5 6 11 6 6 11 8 6 5 9 11 10 6 6 6 6 6 |
| **case3** | 56 | 5 | 3 3 9 1 18 |
| **case4** | 72 | 1 | 0 |
| **case5** | 76 | 1 | 10 |
| **case6** | 87 | 16 | 0 6 6 9 12 12 9 9 5 3 2 1 0 0 0 0 |

(1) Effect of MAXIMUM_SIMULATED_PI on output pattern accuracy

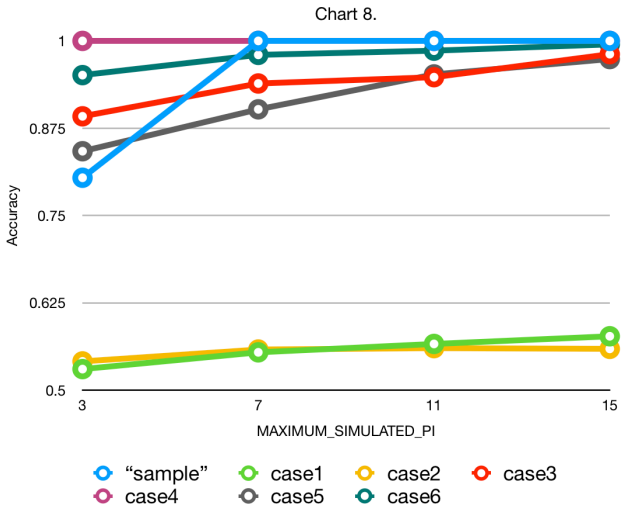
As discussed in step (2) in the algorithm overview section, we select MAXIMUM_SIMULATED_PI PIs with highest dependency for each PO, enumerate all possible input patterns for these PIs, and treat other PIs as don’t-care terms. For each case, we ran the algorithm with MAXIMUM_SIMULATED_PI being 3,7,11, and 15 and for each set-up, the algorithm was executed 3 times. The average accuracy for each case is shown in Table 7 and Chart 8.

To calculate the accuracy, we first generate 5000 random input pattern and simulate them by both the generator and the current circuit. Then, we compare the outputs of both to calculate the accuracy. The simulation program in this part is modified from the work of 王皓仁.

Table 7.

| MAXIMUM_SIMULATED_PI | “sample” | case1 | case2 | case3 | case4 | case5 | case6 |
|---|---|---|---|---|---|---|---|
| **3** | 0.804 | 0.530 | 0.541 | 0.892 | 1.00 | 0.842 | 0.951 |
| **7** | 1.00 | 0.554 | 0.558 | 0.939 | 1.00 | 0.902 | 0.980 |
| **11** | 1.00 | 0.566 | 0.560 | 0.948 | 1.00 | 0.952 | 0.986 |
| **15** | 1.00 | 0.577 | 0.559 | 0.981 | 1.00 | 0.974 | 0.995 |

Chart 8.




For the case “sample”, since there are only 6 PIs (can be seen in Table 6), we always have 1.0 accuracy when MAXIMUM_SIMULATED_PI is greater than or equal to 6.

For case1, we have poor accuracy. This is because more than half of the POs have number of dependent PIs greater than 20. For these POs, we only searched a fairly small portion of the input

boolean space and thus resulted in poor accuracy. Notice that the volume of boolean space grows exponentially with the number of variables. A boolean space would be way beyond our reach even if the number of dependent PIs is only slightly larger than MAXIMUM_SIMULATED_PI.

For case2, we can see from Table6 that all POs have a small number of dependent PIs and therefore we should expect high accuracy after simulation. However, the result is still poor. This is because the number of PIs for the entire circuit is really large (121). Notice that in step (1) of the algorithm, we randomly generate input patterns to find dependent PIs. Since the input boolean space for the entire circuit in this case is extremely large, the algorithm did a poor job finding the dependent PIs.

In short, the accuracy is low for both case1 and case2. Case1 have low accuracy since most POs depend on a relatively large number of PIs and thus can not be enumerated. The bottle neck of this case is at step (2) of the algorithm. Case2 have low accuracy since the number of PIs for the entire circuit is extremely large thus the algorithm did a poor job finding the dependent PIs. The bottle neck of this case is at step (1) of the algorithm.

For case3, case5, and case6, the algorithm performed pretty good. The more effort we spend (setting larger MAXIMUM_SIMULATED_PI), the better accuracy we achieve. These cases all have relatively small number of PI for the entire circuit and small number of dependent PIs for each PO.

Case4 is a special case. Though the circuit have 72 PIs, the only PO is almost always zero. However, when all PIs have input 1, the output would be 1. This circuit is very similar to a 72 input NAND gate. Since the output 1 case is extremely hard to find by the random simulation of the algorithm (step (1)), the algorithm interpret it as a PO with no dependent PI and construct a circuit with constant zero output. Nevertheless, the accuracy testing program also performs random testing and therefore were not able to find the unsatisfied case. This is why though the constructed circuit (constant zero) is not equivalent to the generator (72 input NAND), we still have 1.00 testing accuracy.

(2) Effect of the effort spent on finding dependent PIs on accuracy

In step (1) of the algorithm, we simulate the circuit with random generated input patterns to find the dependent PIs for each PO. The more effort we spend on this step, the more dependent PIs can we find and usually leads to higher accuracy. In this part, we are going to discuss the effect of the effort spent on finding dependent PIs on accuracy.

The detailed procedure for this part is demonstrated by the following pseudocode.

```
For each iteration:
        generate (K * number_of_PI) pairs of random pattern
        simulate and observe PI-PO dependency
        if no new dependent PI is found for all PO in PATIENT iterations:
                break
```

K and PATIENT are adjustable parameters. Increasing either of them can make us find more dependent PIs and usually leads to higher accuracy. However, it would also cost more time. Through some experiments, we found that under the same amount of time consumption, smaller K and larger PATIENT leads to higher accuracy than larger K and smaller PATIENT. In the following, we used case3 as an example and test the accuracy and time used for different parameter set-ups. For Table 9, small K and large PATIENT are used and for Table 10, large K and small PATIENT are used. 3 tests are performed for each set-up. As we can see in the tables, small K and large PATIENT have higher accuracy while both set-ups take about the same amount of time.

On the other hand, if we fix K at 2^3 and adjust PATIENT to different value, we can observe the dependency of accuracy on PATIENT. For each set-up, three tests are performed on case3. The results are shown in Table 11. As we can see, accuracy increases when PATIENT increases. But it quickly saturates when PATIENT exceeds 5.

In this section, we only demonstrate the relationship of the effort spent on finding dependent PIs and accuracy using case3 as an example. However, the results for other cases are about the same.

Table 9.

| Accuracy | Time used (s) |
|---|---|
| 0.987 | 6 |
| 0.994 | 6 |
| 0.994 | 6 |

K=2^3, PATIENT=15

Table 10.

| Accuracy | Time used (s) |
|---|---|
| 0.959 | 6 |
| 0.905 | 6 |
| 0.976 | 6 |

K=2^15, PATIENT=3

Table 11.

| PATIENT | 1 | 2 | 3 | 5 | 10 |
|---|---|---|---|---|---|
| Average accuracy | 0.917 | 0.952 | 0.969 | 0.992 | 0.991 |

(3) Circuit simplification

Step (4) and (6) of the algorithm are circuit simplification. It is performed by a combination of instructions provided by ABC system. During these two steps, the program should reduce the number of gates in the circuit without changing the functionality.

The algorithm simplifies the circuit both individually and jointly for several reasons. First, simplifying the circuit individually takes less time since there are less gates in the circuit when performing the task. Thus, individual circuit simplification are used. Second, after combining the circuits into a multi-PO circuit, many gates can be shared. Thus, joint simplification are used. Third, when performing simplification, the improvement saturates quickly. For example, the first simplification step may reduce the circuit by 7% but the fifth may only reduce the size by less than 1%. We can view this situation as sticking in a local optima. Performing the circuit simplification in a two-staged manner gives us a second chance to further reduce the circuit size effectively.

Due to the reasons mentioned above, the algorithm was designed to perform two-staged circuit simplification. However, we later found out that performing joint simplification would change the functionality of the circuit for some unknown reasons. The joint simplification step are therefore removed from the algorithm. All the results described previously is performed “without” the joint circuit simplification step. We performed an experiment on case3. We construct the circuit under four different settings: no simplification, individual simplification only, joint simplification only, and both. For each setting, we ran the program three times and record the circuit size and testing accuracy. The average circuit size and accuracy for each settings are shown in Table 12.

Table 12.

| | No simplification | Individual simplification only | Joint simplification only | Both |
|---|---|---|---|---|
| Average circuit size | 11241 | 9608 | 9781 | 6950 |
| Average accuracy | 0.975 | 0.968 | 0.650 | 0.651 |

As we can see in the table, performing either individual or joint circuit simplification reduced the original circuit size by about 14%. Performing both reduced the number of gates by about 38%. However, whenever the joint simplification step was performed, the functionality of the circuit somehow changed and the accuracy were reduced significantly. A lot of time were spent on figuring out the reason of this result but we still do not have any conclusion yet. Perhaps what the ABC system really performed was not what we thought.

# 4. Conclusion

In the contest, 0.9999 accuracy is required. In our experiments, most cases can not reach such high accuracy. However, the time limit for each case is one hour in the contest while the algorithm we discussed takes only a few seconds to a few minutes. By adjusting the parameters in the algorithm, such as increasing MAXIMUM_SIMULATED_PI, K, and PATIENT, we may achieve 0.9999 accuracy within the one-hour time limit for some cases, cases like case3, case4, case5, and case6, for example. But it is clear that the current algorithm design is still incapable of dealing with cases like case1 and case2. Better approaches still need to be developed to put our work into industrial use.

# 5. Future Work

(1) Better ways to identify dependent PIs

The accuracy of the constructed circuit depends largely on how many dependent PIs can be identified. Missing even a few dependent PIs could reduce the accuracy significantly. Random simulation (step (1) of the algorithm) can be very effective at first, but the marginal improvement reduces quickly. A smarter way need to be developed to identify dependent PIs effectively, especially when dealing with cases like case2 which the number of PI is large.

(2) Better ways to construct the circuit after dependent PIs are found

Even if we are able to find all dependent PIs for all POs, if the number of dependent PIs are too large, the current algorithm are still incapable of handling it since it is impossible to enumerate them (step (2) of the algorithm). A smarter way need to be developed to construct the circuit, especially when dealing with cases like case1 which the number of dependent PIs for POs are large. Perhaps some heuristics can be used.

One possible way is to guess the unknown patterns via looking at similar patterns. Take Table 13 as an example. Let us say the Karnaugh map for a PO is as Table 13. One corresponding value for a given input pattern is still unknown (denoted as question mark in the table). It is pretty intuitive to assume the missing output pattern is 1 since most its neighbor have output value 1.

We can interpret this approach by two ways. First, to find clusters in the boolean space. Second, to interpret the missing term as don't-care term and to reduce simplify the circuit as much as possible. Notice that if we treat this missing term as don't care term and try to simplify the circuit, we still get output one for this term.

Table 13

| | PI_1=0, PI_2=0 | PI_1=0, PI_2=1 | PI_1=1, PI_2=1 | PI_1=1, PI_2=0 |
|---|---|---|---|---|
| PI_3=0, PI_4=0 | 0 | 0 | 1 | 1 |
| PI_3=0, PI_4=1 | 0 | 0 | ? | 1 |
| PI_3=1, PI_4=1 | 0 | 0 | 1 | 1 |
| PI_3=1, PI_4=0 | 0 | 0 | 0 | 0 |

Another possible way is to use the concept of conditional don't-care terms. Let us say X=a * f1(b, c, d, e) + a' * f2(g, h, i, j), where f1 and f2 denotes some boolean functions. X is a boolean function with 9 variables. But in fact, b, c, d, and e becomes don't-care when a is 0 and g, h, i, and j becomes don't-care when a is 1. Therefore, we only have to find two 4-variable boolean functions instead of one 9-variable boolean function. This leads to significant reduction of searching time.

Moreover, this kind of situation is actually quite common in practice. Variable a is like a switch (a MUX) in the circuit selecting the desired boolean function. Figure 14 is a plot of CPU architecture, the arrows denotes control signals. If we can find all these control signals, we can reduce the functional complexity enormously.

So perhaps before step (1) of the algorithm, we can do a quick test to find if there is any variable that, when assigned, can produce a lot of conditional don't-care variables. This can even been done recursively by observing the dependency of POs on PIs after multiple variable assignments.

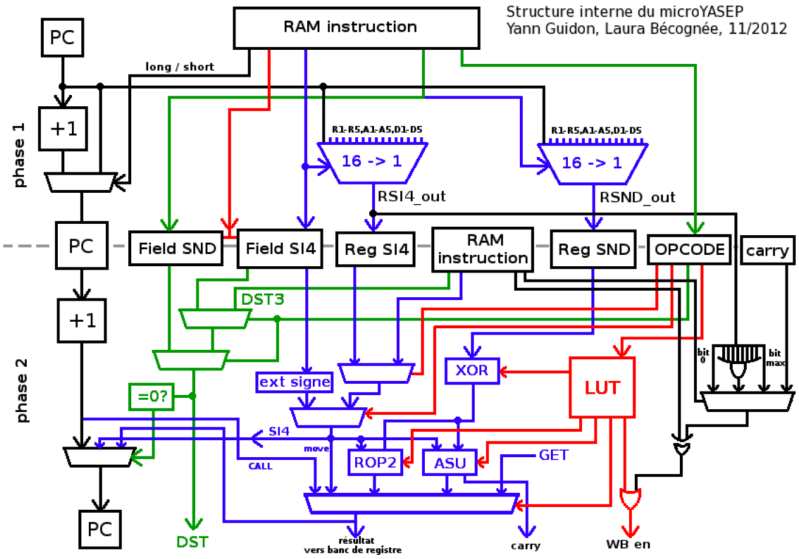



**Figure 14.**

(3) Case-dependent parameter setting

In the current work, all parameters such as MAXIMUM_SIMULATED_PI, K, and PATIENT, must be determined beforehand. However, since each case differs a lot, it is better that our algorithm can self-adjust its parameters when dealing with different cases. For example, case1 have large number of dependent PIs for POs and need a larger MAXIMUM_SIMULATED_PI. Case2 have a lot of input variables and should be treated with larger K, and PATIENT, to find the dependent PIs.

# 6. Best Results for Each Given Test Case

The final version have parameter setting as follows:

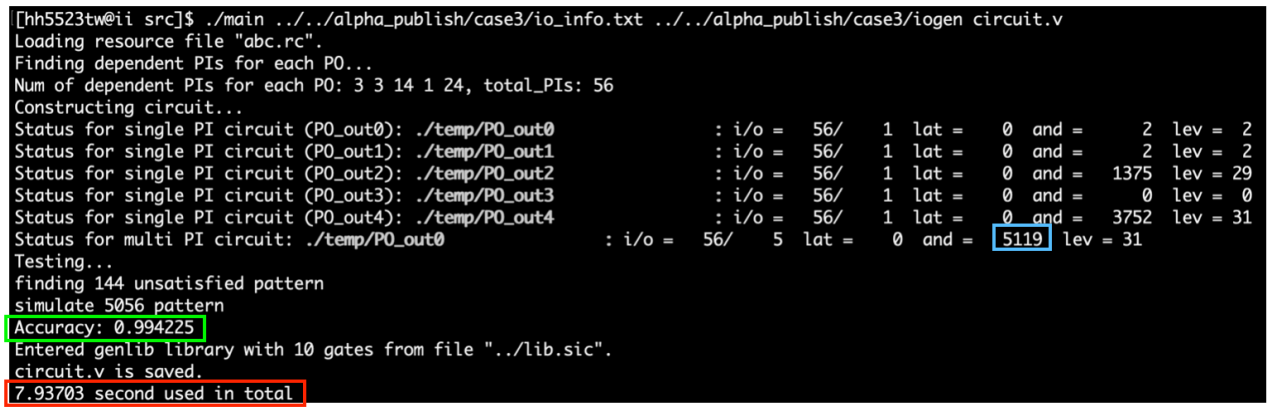
MAXIMUM_SIMULATED_PI ==7
K ==3
PATIENT==30

The results for each sample cases under this setting are shown in Table 15.

Table 15

| Samle name | Accuracy | Number of gates | Time used (s) |
|---|---|---|---|
| **“sample”** | 1 | 15 | 0.19 |
| **case1** | 0.573 | 116828 | 39.47 |
| **case2** | 0.563 | 11407 | 98.07 |
| **case3** | 0.994 | 5119 | 7.93 |
| **case4** | 1 | 0 | 1.55 |
| **case5** | 0.994 | 3875 | 3.91 |
| **case6** | 0.997 | 3257 | 23.98 |

# 7. Results Interpretation




Number of gates Accuracy Time used

# 8. References

1.http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/links/espresso.5.html
2.https://people.eecs.berkeley.edu/~alanmi/abc/