Project Demo

EE658: Diagnosis and Design of Reliable Digital Systems
Team 12

Wei-Chih, Lin, wlin3053@usc.edu
Tsai-Ning, Lien, tsaining@usc.edu
Lung-Hao, Tsai, lunghaot@usc.edu

Group Project Responsibilities

To be completed before the demo

Name	List of Tasks
Wei-Chih, Lin	PFS, RFL, RTPG, Logicsim, SCOAP, TPG(D-Frontier)
Tsai-Ning, Lien	DFS, PODEM, Logicsim, TPG(Fault Order, J-Frontier, v1)
Lung-Hao, Tsai	DALG, PODEM, Logicsim, OOP, TPG(baseline/v0, v2)

Group Project Responsibilities

To be completed before the demo

Task	Names
LSIM P1	Lung-Hao Tsai, Wei-Chih Lin, Tsai-Ning Lien
LSIM P2	Lung-Hao Tsai, Wei-Chih Lin, Tsai-Ning Lien
RTPG	Wei-Chih Lin
RFL	Wei-Chih Lin
PFS	Wei-Chih Lin
DFS	Tsai-Ning Lien
SCOAP	Wei-Chih Lin, Tsai-Ning Lien
DALG	Lung-Hao Tsai, Tsai-Ning Lien
PODEM	Tsai-Ning Lien, Lung-Hao Tsai, Wei-Chih Lin
ATPG	Lung-Hao Tsai
OOP / Structures	Lung-Hao Tsai
TPG (v0/baseline)	Lung-Hao Tsai
TPG (v1,v2)	Lung-Hao Tsai, Tsai-Ning Lien
TPG (D-Frontier)	Wei-Chih Lin
TPG (Fault Order)	Tsai-Ning Lien
TPG (J-Frontier)	Tsai-Ning Lien

Teamwork
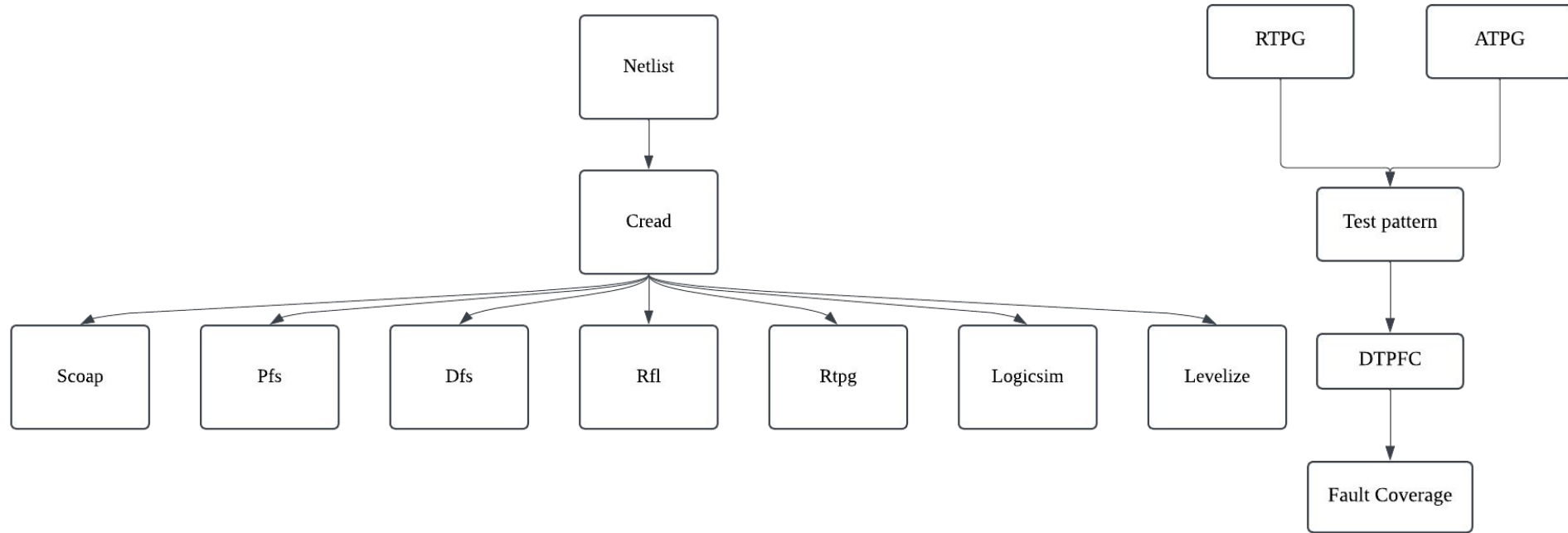*Let's be more professional!*

- Use draw.io for your schematic, flowcharts, and all other figures.
- Task assignment: monday.com
- GitHub

Overall Flow Chart for TPG

Let's discuss our Design and Algorithm

ATPG (D-Algorithm)

- Main function void dalg()
- Recursive function bool my_dalg(node_class *current_np,frontier &current_Frontier)
- Sub-functions

  - bool imply_and_check(node_class *j, int v_j,frontier &current_Frontier)

- The main function dalg calls the recursive my_dalg with the variable of current_np, and current_Frontier to make sure the function into the stack is their local frontier.

- from the beginning of the my_dalg, we calls the imply_and_check, to make sure the value we assign from D-frontier of J-frontier is correct or not. (using stack to push the node that need to be deal with)

ATPG (PODEM)

- Main function void podem()
- Recursive function bool podem_sub(node_class *d_node,frontier &current_Frontier)
- Sub-functions

    - pair<node_class*, int> objective(node_class * np,int fault_value,frontier current_Frontier)

    - pair<node_class*, int> backtrace(node_class *k, int v_k)

    - void imply(node_class *j, int v_j,frontier &current_Frontier)

    - bool activatedFault(node_class* fault_node)

    - bool existsXPathToPO(node_class* fault_node)

    - bool testImpossible(node_class *d_node)

The main function podem calls the recursive podem_sub function. podem_sub relies on objective to select a target node and value, backtrace to find a controllable PI, and imply to propagate values. Testability validation in podem_sub is handled by testImpossible, which internally uses activatedFault and existsXPathToPO.
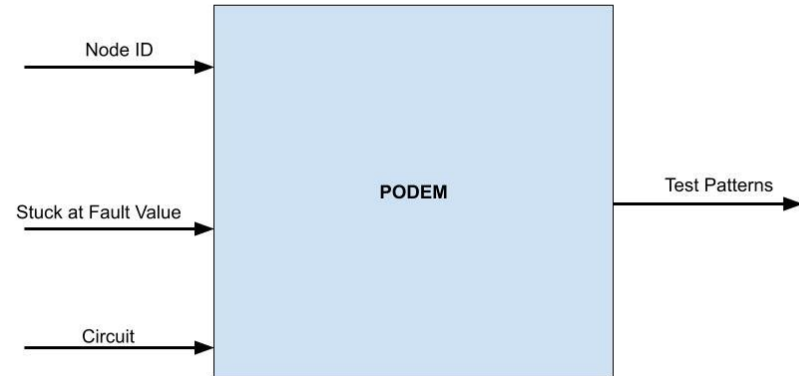
ATPG (DALG, PODEM, single fault)

Where are you standing right now?

- Explain in your own words, but address these following questions.
  - c17 → All good with DALG and PODEM? Yes, because the circuit is not large, we can find the test patterns easily.
  - All other mini-circuits? The same concept, All good for mini circuits
- Draw a block diagram of your DALG / PODEM module.
  - *What is input and output?*
- How to check the correctness of the solution in the previous modules?
  - Run PFS, DFS after ATPG (and manual calculation) (on next page)
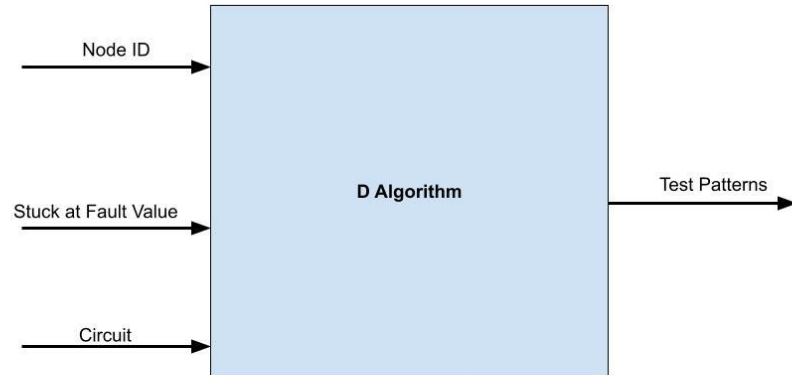
D Algorithm Module

PODEM Module

ATPG (DALG, PODEM, single fault)

What about large circuits?

- Why the TA did not provide the golden results for the large circuits?

  For any given fault, many input combinations can detect it. Providing golden results might suggest only one "correct" pattern, which isn't true. (Enumerating all possible patterns would make the golden result file too large for large circuits.)

- Draw a modular block diagram on how to test correctness of your DALG / PODEM solutions.
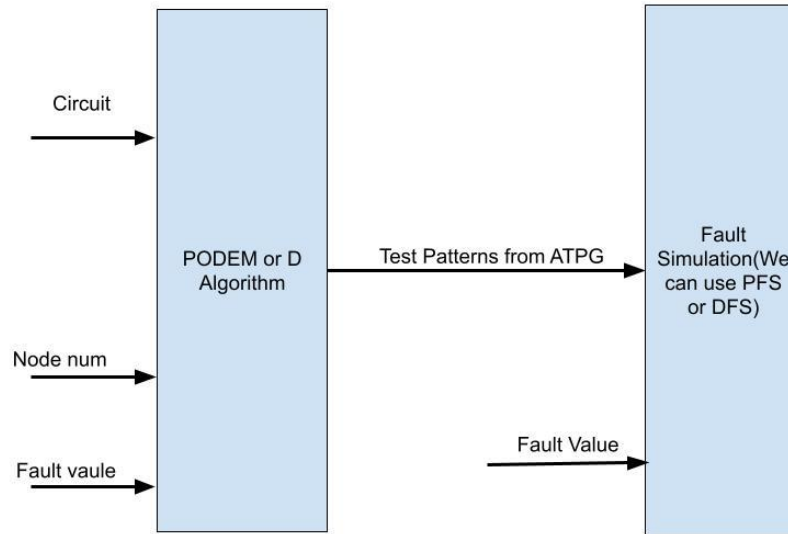
Test Pattern Generation (TPG)

What does it exactly mean?

- Problem statement?
  The problem statements for TPG involve generating test patterns for digital circuits. Our goal is to detect the stuck at fault in the circuit with the highest possible fault coverage, minimize runtime and test volume, without significant concern for memory or cost for this specific project phase.

- How to compare results (criteria, objective)

  1. Execution Time: We can compare the execution time with same circuit, same ATPG algorithms but with different versions in the same heuristic topic. The fewer execution time, the better heuristics.

  2. Fault Coverage: We can compare the fault coverages. The higher fault coverage, the better heuristics.

  3. Test Volume: We can compare the number of test volumes(patterns) to detect the fault that can be found by the test pattern we generate. The fewer test patterns, the better heuristics.

  4. Memory Usage

  5. Cost

- **Command:** tpg <-rtp version fault_coverage> <-df version> <-jf version> <ATPG(D Algorithm or PODEM) > <tp_result.txt>

DTPFC(Test Pattern Fault Coverage)

What does it exactly mean?

- **Description:** DTPFC inputs external test pattern file and runs fault simulation. Finally, we get fault coverage for these patterns. We have a integer number **freq,** which is the sample frequency of test patterns to generate fault coverage result.

- **Design**:

Algorithm 1 DTPFC

Input: TP: External Test Pattern File
Output: FC: Fault Coverage
while TP is not empty **do**
    $tp \leftarrow$ select Test Patterns from TP
    $fl \leftarrow DFS(tp)$
    $fc \leftarrow fl/total faults$
    $FCs \leftarrow FCs + fc$
end while

- **Command:** tpfc <tp_fname> <freq> <fctpfc_report_fname>

Our Testing Criteria

We chose **four** heuristics: **Transition from RTPG to ATPG**, **Fault Order**, **J-frontier selection**, and **D-frontier selection**.

When testing a single heuristic, comparisons will only be made within the same heuristic, under the same ATPG, but between different versions. **For example**, in the case of "Transition from RTPG to ATPG," we have PODEM: v0, v1, v2 and D-ALG: v0, v1, v2 respectively.

For the other heuristics, we will only run -rtp v0(Baseline) and switch between different versions within that heuristic, also testing them with PODEM and D-ALG respectively.

As for the circuits, we choose **c17**, **c2,** and **c880**.

Finally, we will test the combination we assume achieves the smallest execution time and compare it against the baseline.

Description: This version of RTPG is the baseline design. Before achieving the max FC or FL is not empty, we run ATPG to get test patterns and use DFS as a fault simulation. If we achieve the max FC or FL is empty, stop RTPG v0.

Design:

Algorithm 1: Vanilla test pattern generate (baseline)

Input: circuit, choice of ATPG (DALG, PODEM)
Output: TPs: A set of test patterns
$FL \leftarrow$ all single stuck at faults
$TPs \leftarrow$ empty list
while *FL is not empty or max FC is achieved* **do**
> $f \leftarrow$ select random from FL
> $tp \leftarrow ATPG(f)$
> tp is ternary, you may need to modify it here
> $TPs \leftarrow TPs + tp$
> $DFs \leftarrow PFS(FL, tp)$ (DFs: list of detected faults)
> $FL \leftarrow FL - DFs$

return TPs

Command: tpg -rtp v0 fault_coverage_number ATPG(PODEM or D ALG) tp_output.txt

Results: Baseline Design+DFS

D Algorithm

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.04	8	100
c2	0.08	12	88.6
c880	53.1	168	100

PODEM

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.046	9	100
c2	0.078	11	88.6
c880	50.2	168	100

Analysis:

Compare with these two ATPG Algorithm, both runtime, test volume and fault coverage don't change a lot. It means these 2 different ATPG algorithms don't play important roles on this <RTPG-v0>.

Description: Start with RTPG, run DFS for each new test pattern until the fault coverage reaches a given value

Design:

```
Algorithm 1 Run RTPG
  Input: Circuit, Run RTPG
  Output: TPs: A set of test patterns
  FL ← all single stuck at faults
  TPs ← empty list
  while MaxFC not reach or FL is not empty  do
    f ← select random from FL
    tp ← RTPG(f)
    tp is ternary, you may need to modify it here
    TPs ← TPs + tp
    DFs ← DFS(tp)
    FL ← FL − DFs
  end while
  return  TPs
```

Command: tpg -rtp v1 fault_coverage ATPG(D Algorithm or PODEM) tp_output.txt

Results: RTPG+DFS

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.022	52	100
c2	2.36	1052	88.6
c880	90	10000	99.83

Analysis:

Because we use RTPG only, it takes more time and more test patterns to achieve the ideal fault coverage.

The biggest difference is c880 circuit. Right now, it is only 99.83% although we run 10000 patterns. Random resistant pattern happens. We cannot get 100% fault coverage ideally.

Description: Begin with ATPG and continue until the improvement in fault coverage (delta FC) is below a specified value.

Design:

Algorithm 1 Run ATPG with Delta FC

Input: Circuit, choice of ATPG (DALG, PODEM)
Output: TPs: A set of test patterns
$FL \leftarrow$ all single stuck at faults
$TPs \leftarrow$ empty list
while $DeltaFC$ not reach **do**
   $f \leftarrow$ select random from FL
   $tp \leftarrow ATPG(f)$
   tp is ternary, you may need to modify it here
   $TPs \leftarrow TPs + tp$
   $DFs \leftarrow DFS(tp)$
   $FL \leftarrow FL - DFs$
   $NewFC \leftarrow tpfc$
   $DeltaFC \leftarrow NewFC - OldFC$
end while
return TPs

Command: tpg -rtp v0 fault_coverage_number ATPG(PODEM or D ALG)   tp_output.txt
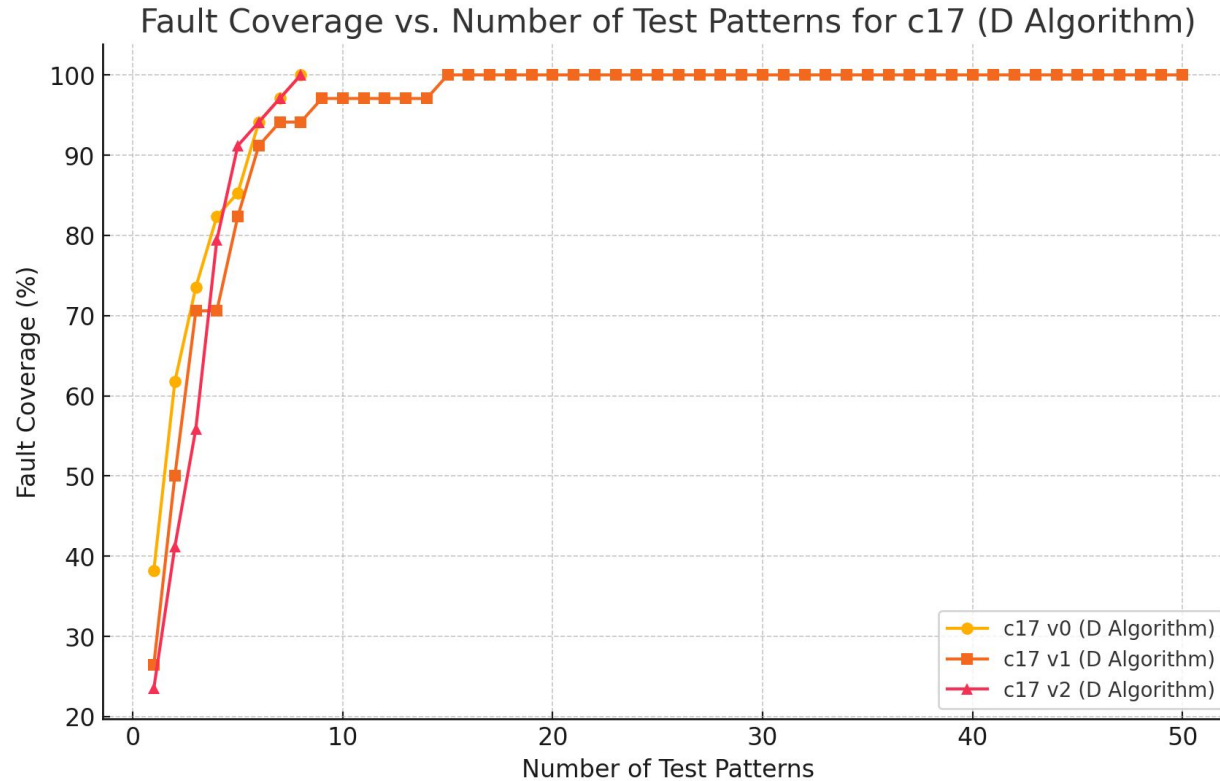
Results: ATPG+DFS with DeltaFC=5

D Algorithm

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.002	8	100
c2	0.03	3	47.73
c880	0.13	4	46

PODEM

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.007	9	100
c2	0.07	5	68.3
c880	0.26	4	34.12

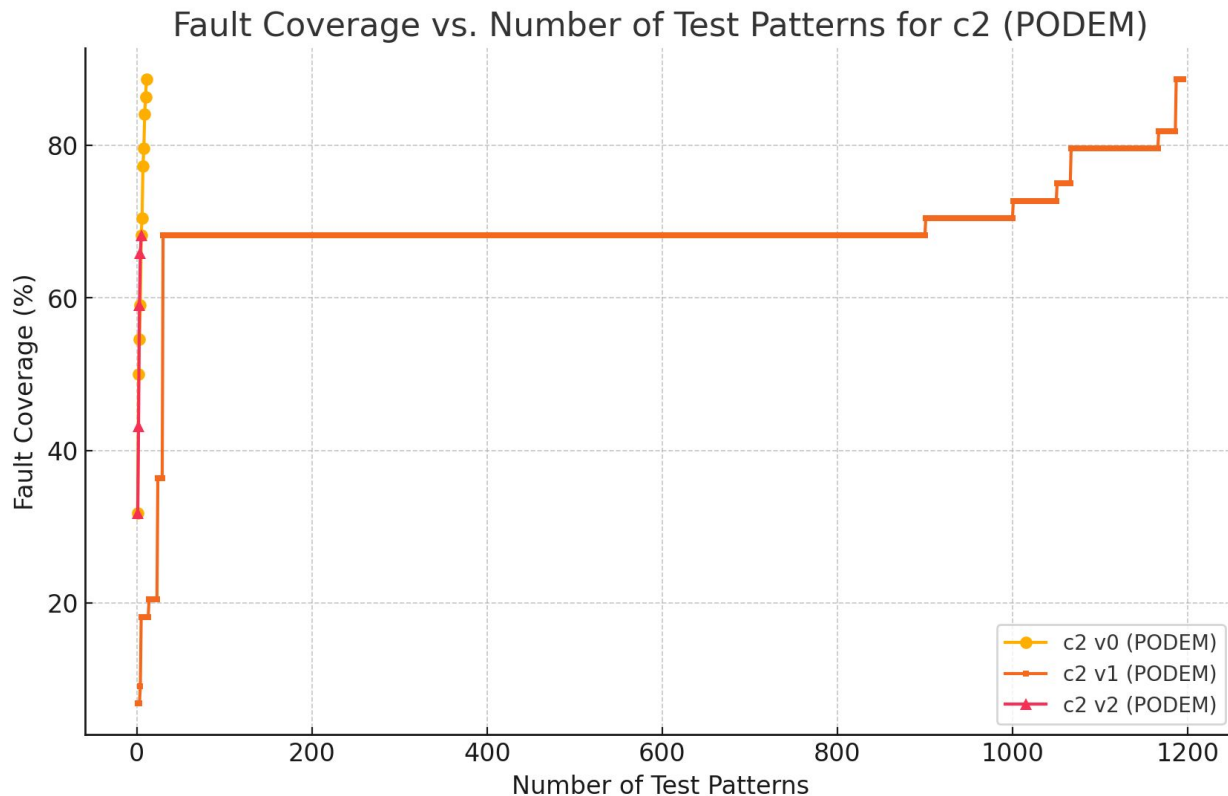**Analysis:**

We can found that because delta FC is 5 so the FC cannot reach max FC.Also, in our code, D Algorithm has better FC and less runtime.
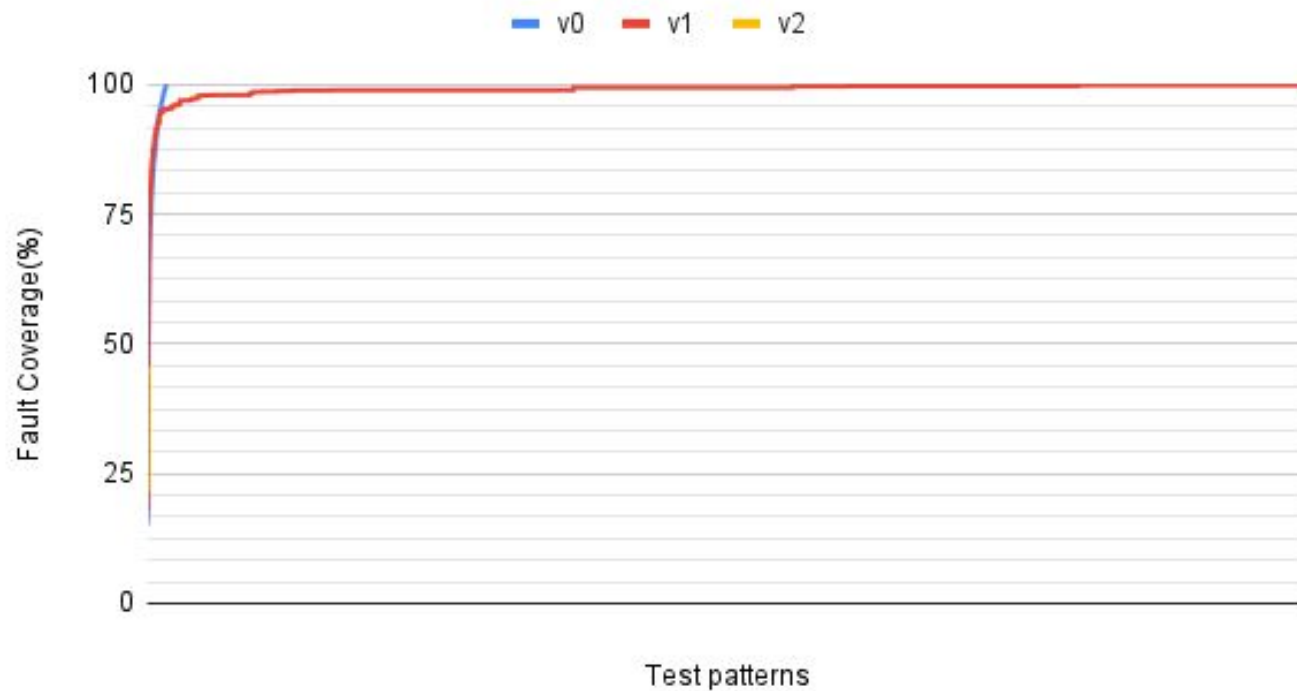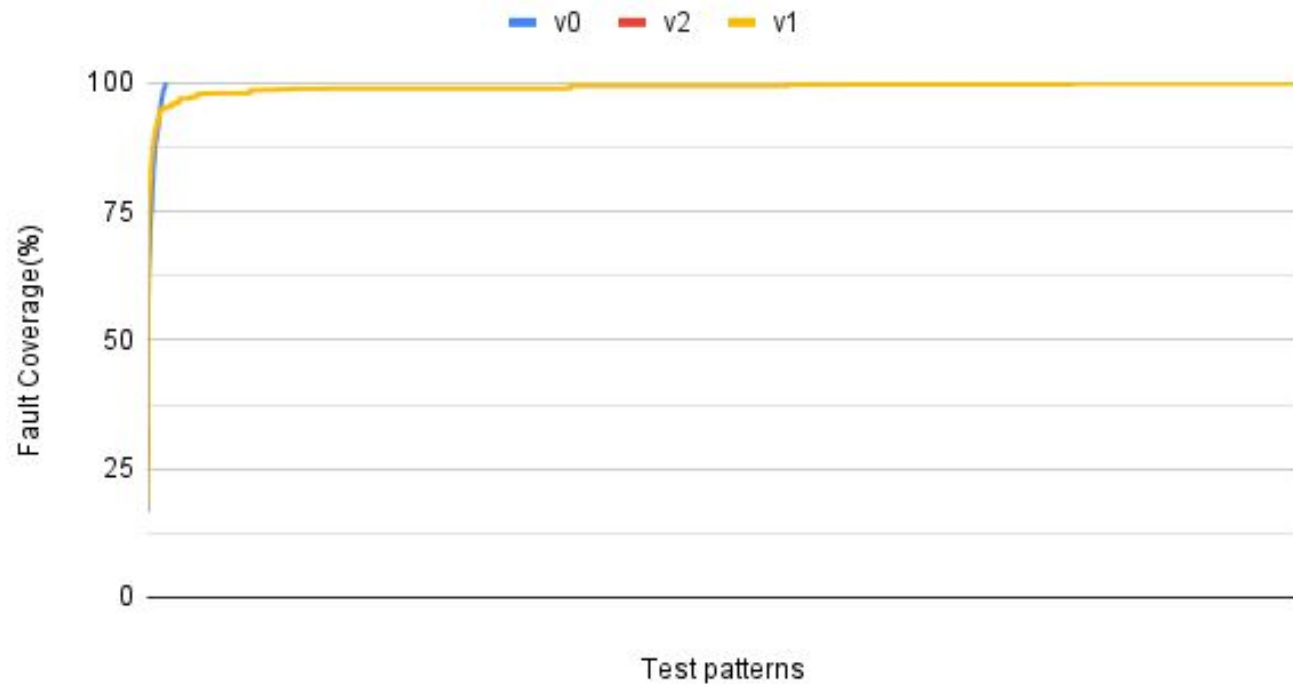
Fault Coverage vs. Number of Test Patterns for c17 (D Algorithm)

Fault Coverage vs. Number of Test Patterns for c17 (PODEM)

Fault Coverage vs. Number of Test Patterns for c2 (D Algorithm)

Fault Coverage vs. Number of Test Patterns for c2 (PODEM)

c880 D Algorithm

c880 PODEM

Legend: v0, v2, v1

Fault Coverage(%) vs Test patterns
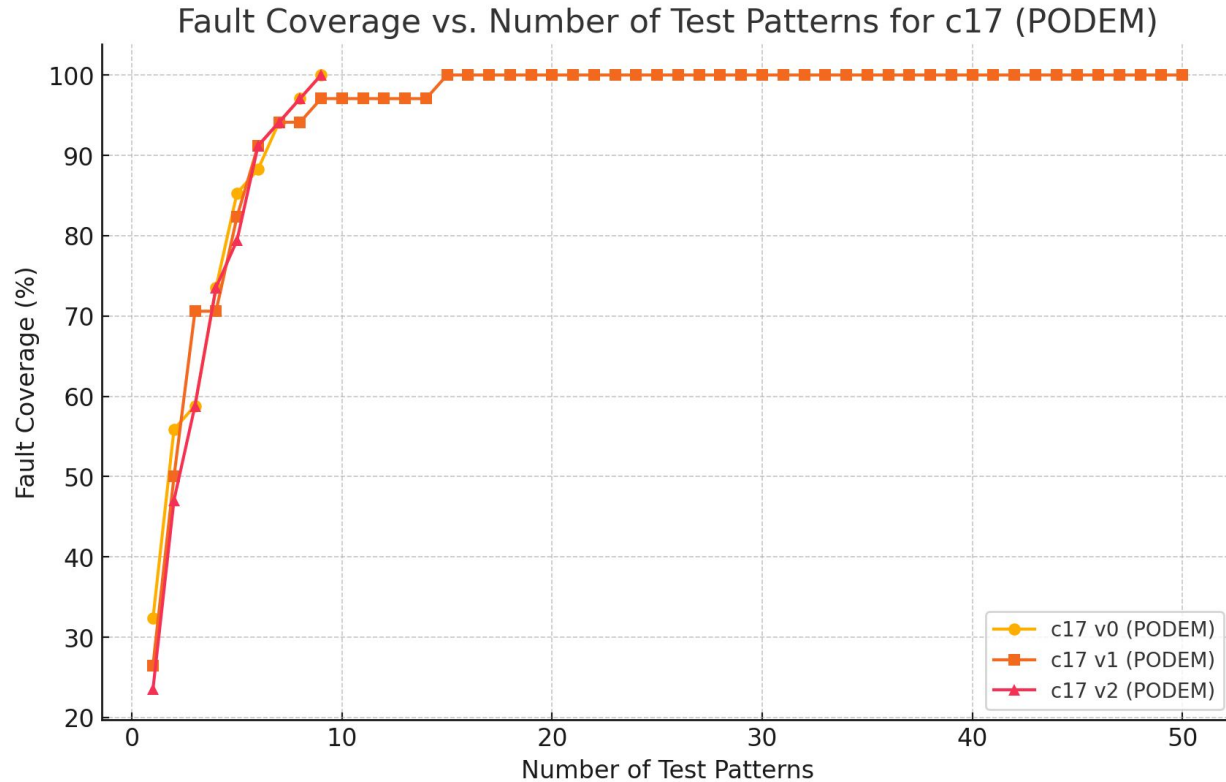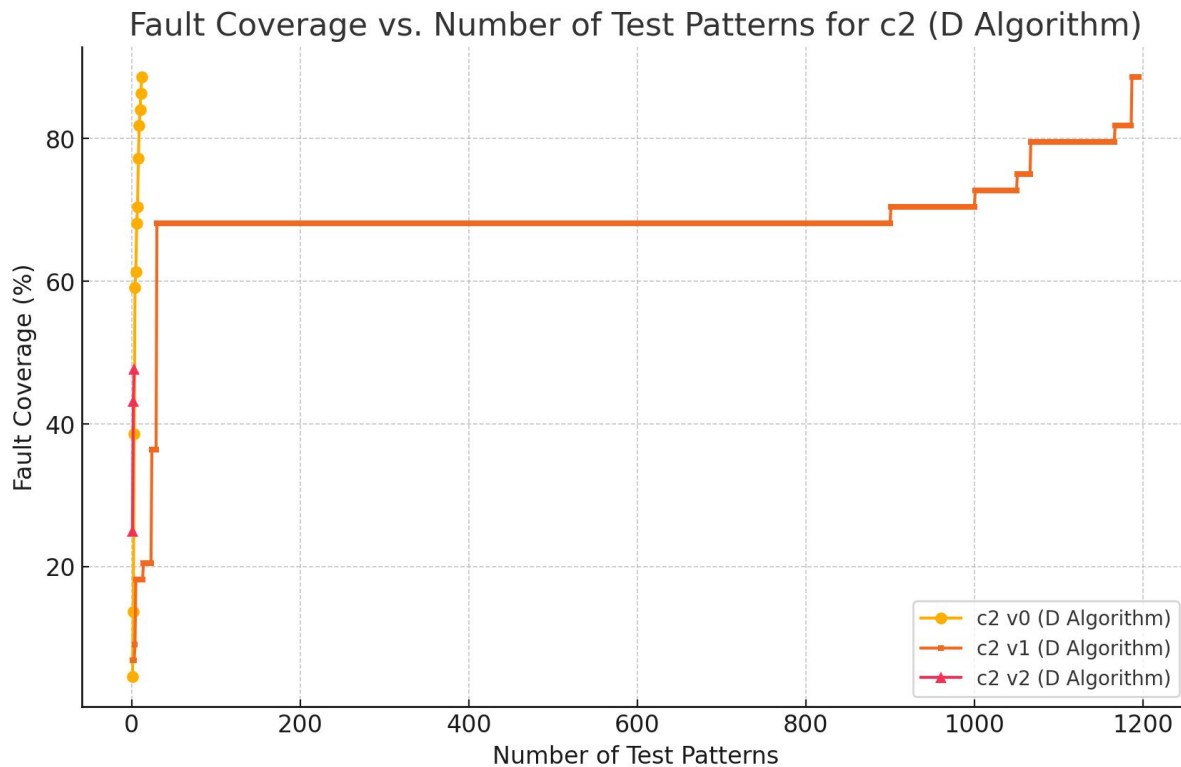
Description: Choose the node from the D-frontier with the lowest/highest number.

Intuition: For PODEM and D ALgorithm, they need to choose D Frontiers to backtrack or backtrace. Hence, I need to compare with v0 to prove that this heuristic can improve any parameter. And we should choose **Lowest node number** because nodes are close to PI usually, which generates test patterns faster.

Design:

```
Algorithm 1 D-Frontier -df nl
    if D-frontier stack is not empty then
        if -df nl then
            for D-frontier stack sorted by number from low to high do
                node = top node from D-frontier stack with lowest level
                remove node with lowest level from D-frontier stack
                Backtrack()
            end for
        end if
    end if
```

Fig. Pseudo Code of -df nl

Design:

Algorithm 1 D-Frontier -df nh

---

   if D-frontier stack is not empty **then**
      if -df nh **then**
         for D-frontier stack sorted by number from high to low **do**
            node = top node from D-frontier stack with highest level
            remove node with highest level from D-frontier stack
            Backtrack()
         end for
      end if
   end if

Fig. Pseudo Codes of -df nh

Command: tpg -rtp v0 fault_coverage_number -df nl ATPG(PODEM or D ALG)   tp_output.txt

        tpg -rtp v0 fault_coverage_number -df nh ATPG(PODEM or D ALG)   tp_output.txt

Results:

D Algorithm

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.045	9	100
c2	0.070	12	88.6
c880	41.55	152	100

PODEM

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.040	8	100
c2	0.060	10	88.6
c880	40.77	150	100

Analysis:

In this case, in small circuit, the runtime doesn't change a lot. But for c880, the time is a little smaller than the runtime in rtpg v0. So this heuristic reduce the runtime.

Description: Select the node from the D-frontier that is at the lowest/highest level.

Intuition: For PODEM and D ALgorithm, they need to choose D Frontiers to backtrack or backtrace. Hence, I need to compare with v0 to prove that this heuristic can improve any parameter. And we should choose **Lowest node level** because nodes are close to PI usually, which generates test patterns faster..

Design:

Algorithm 1 D-Frontier -df ll

---
if D-frontier stack is not empty **then**
    if -df ll **then**
        for D-frontier stack sorted by level from low to high **do**
            node = top node from D-frontier stack with lowest level
            remove node with lowest level from D-frontier stack
            Backtrack()
        end for
    end if
end if

Pseudo Code of -df ll

Design:

Algorithm 1 D-Frontier -df lh

---
if D-frontier stack is not empty **then**
   if -df lh **then**
      for D-frontier stack sorted by level from high to low **do**
         node = top node from D-frontier stack with highest level
         remove node with highest level from D-frontier stack
         Backtrack()
      end for
   end if
end if

Fig. Pseudo Codes of -df lh

Command: tpg -rtp v0 fault_coverage_number -df ll   ATPG(PODEM or D ALG)   tp_output.txt

        tpg -rtp v0 fault_coverage_number -df lh ATPG(PODEM or D ALG)   tp_output.txt

Results:

D Algorithm

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.036	7	100
c2	0.062	12	88.6
c880	41.93	150	100

PODEM

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.033	7	100
c2	0.060	10	88.6
c880	40.98	154	100

Analysis:

In this case, in small circuit, the runtime doesn't change a lot. But for c880, the time is smaller than the runtime in rtpg v0. So this heuristic reduce the runtime.

c17 Algorithm

Legend: II — nI — No D-Frontier Heiristics

Fault Coverage(%) vs Test patterns

D Frontier: c17 in different version of D Frontier

c17 PODEM

Legend: ll, nl, No D-Frontier Heiristics

Y-axis: Fault Coverage(%)
X-axis: Test patterns

c2 D-Algorithm

Legend: ll — nl — No D-Frontier Heiristics

Fault Coverage(%) vs Test patterns

D Frontier: c2 in different version of D Frontier

Page 2/2

c2 PODEM

Legend: ll — nl — No D-Frontier Heuristics

c880 D-Algorithm

Legend: ll — nl — No D-Frontier Heuristics
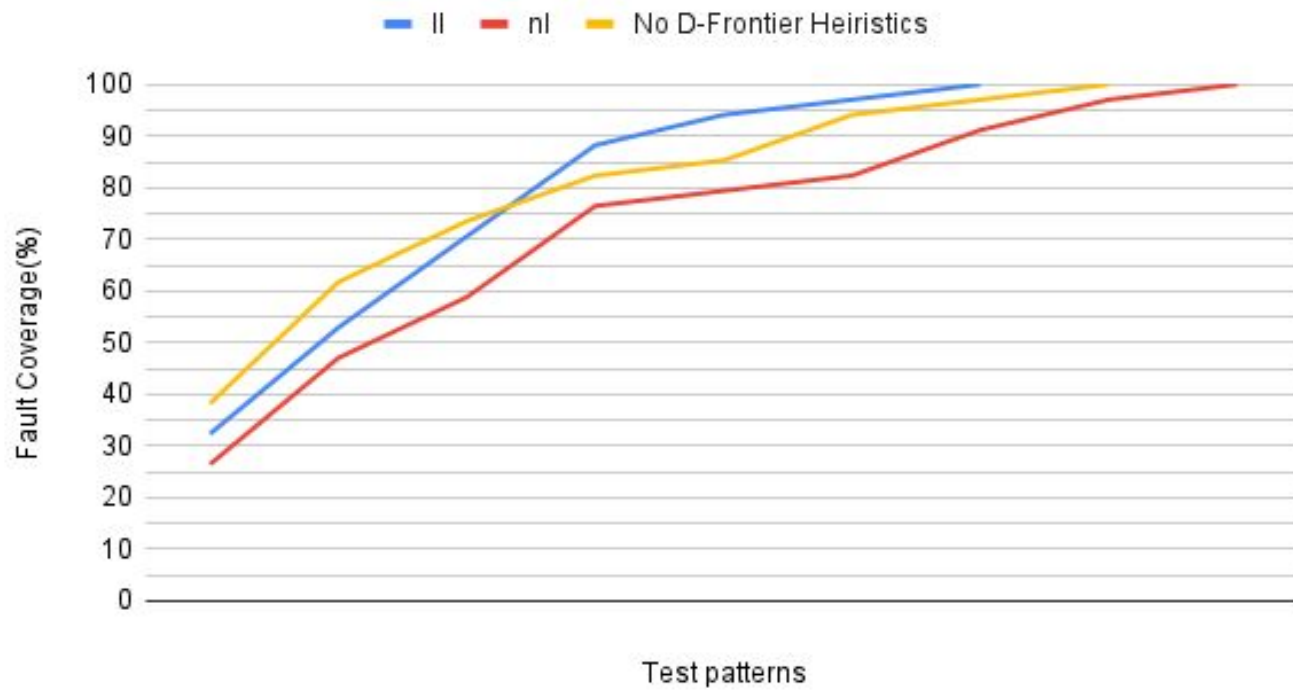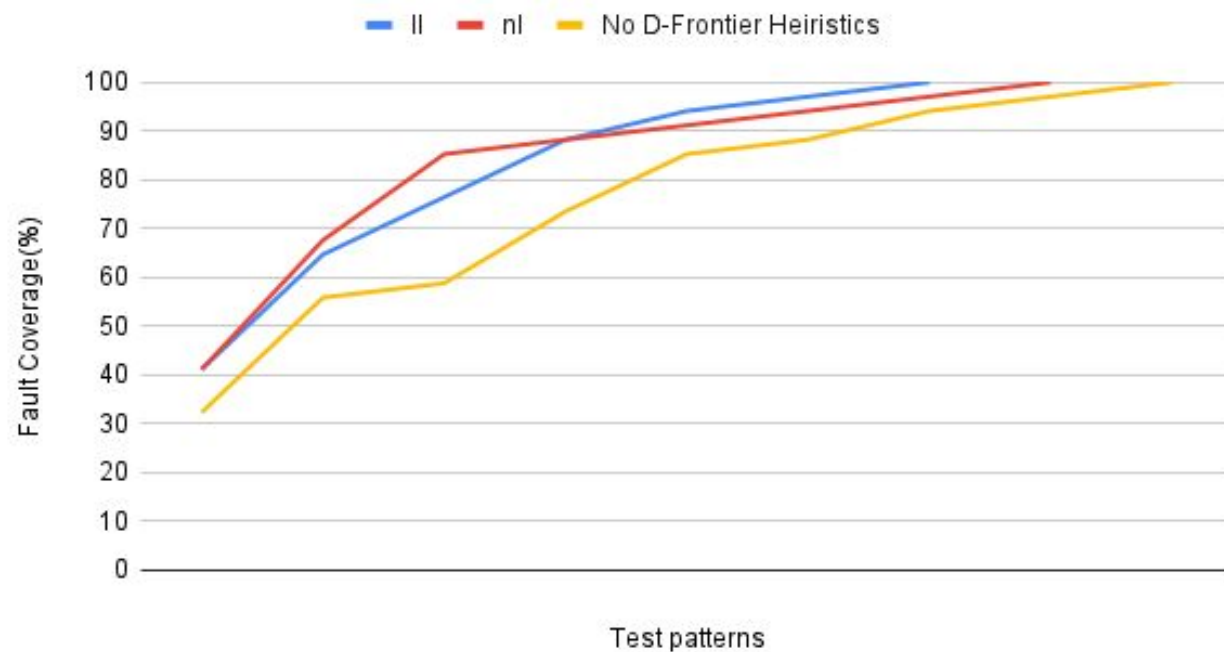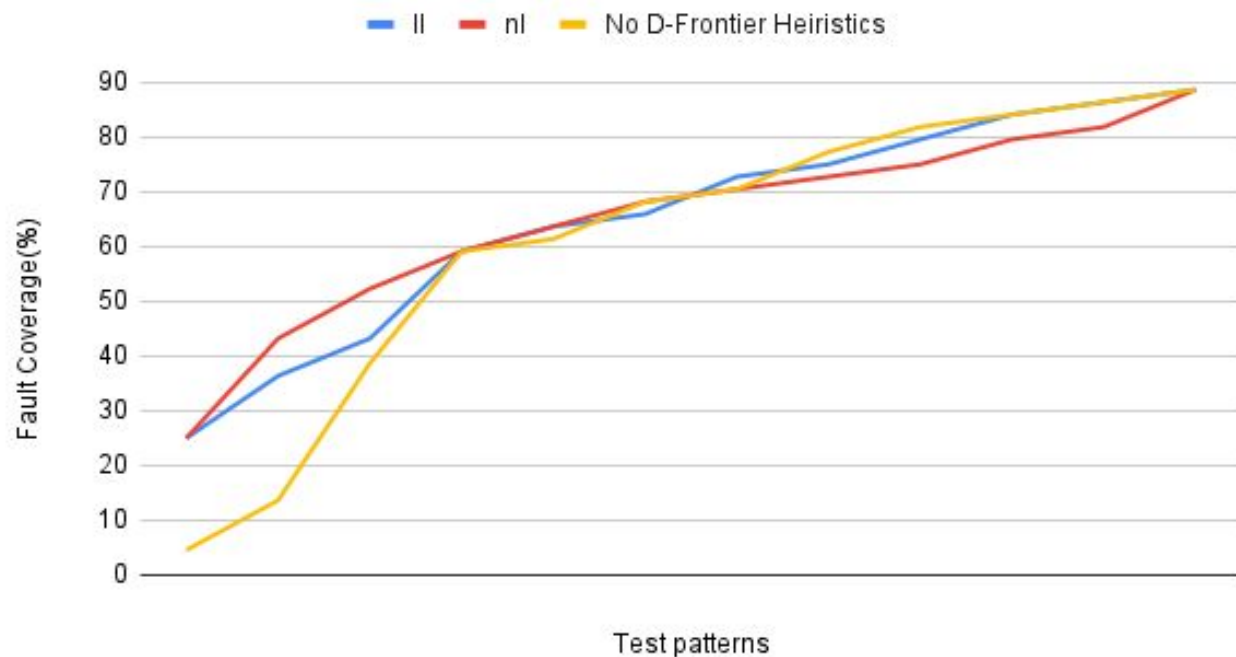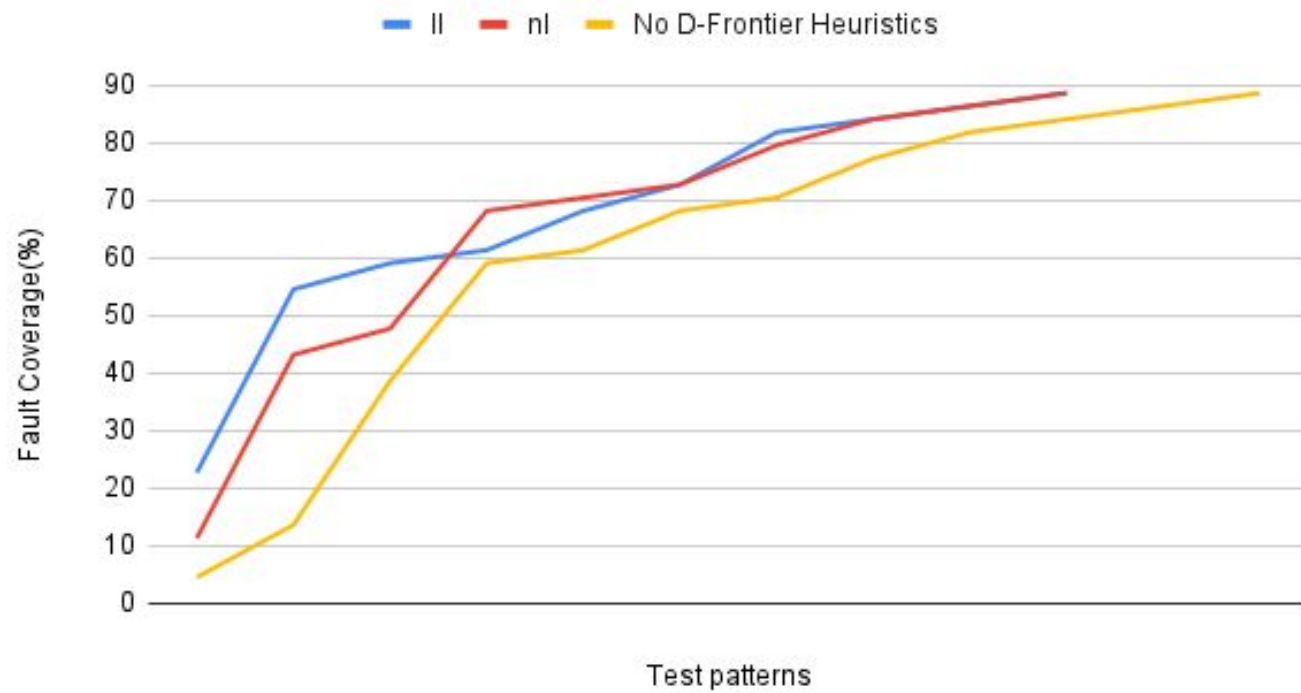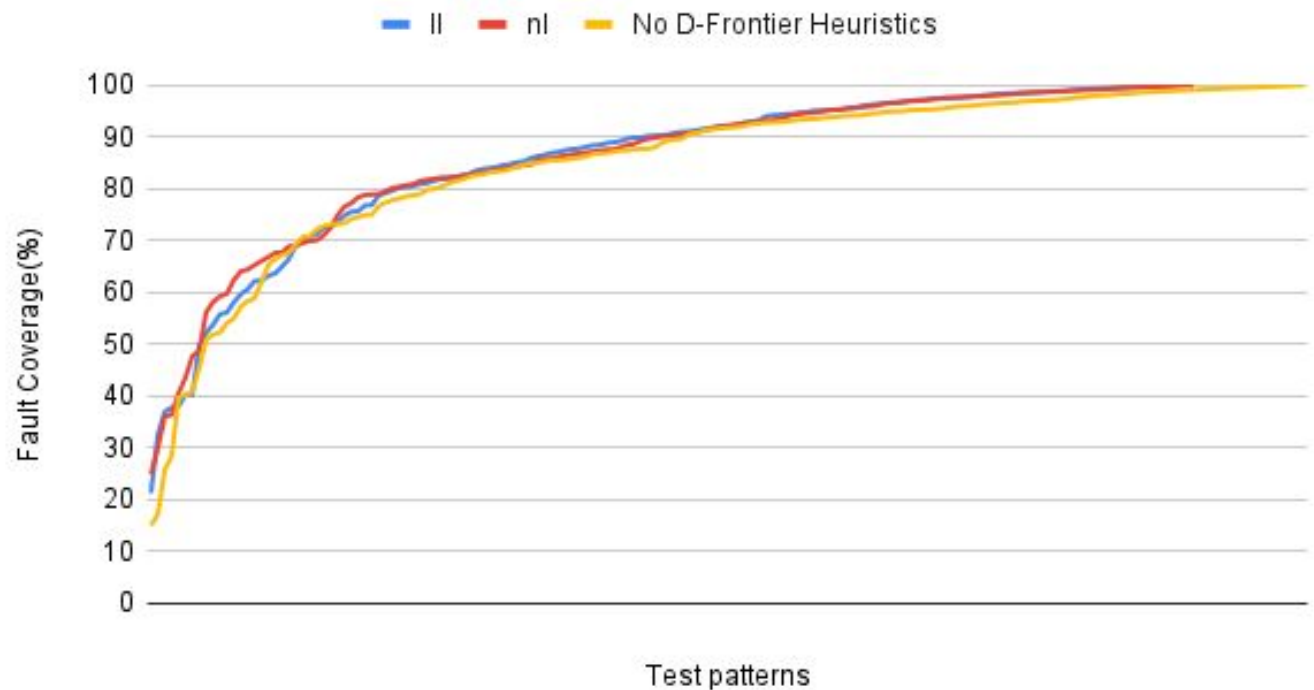
c880 PODEM
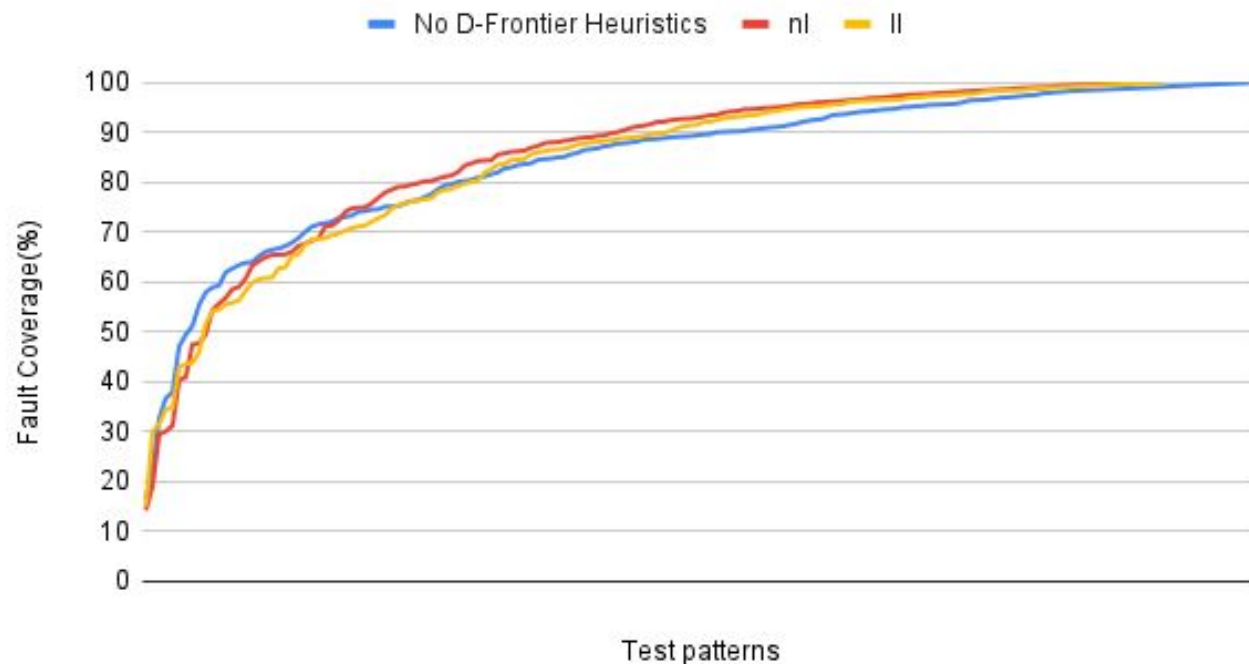
Intuition: This heuristic is based on the idea that inputs with higher controllability values are harder to control, which may lead to longer runtime or more complex computations. By selecting the input line with the lowest SCOAP controllability, we aim to minimize the effort (runtime of our concern) required to justify the output of a logic gate.

Design:

Algorithm 1 J-Frontier -jf v0
___
    if *J*-frontier stack is not empty **then**
        if -jf v0 **then**
            for *J*-frontier stack sorted by controllability **do**
                node = top node from *J*-frontier stack
                remove node from *J*-frontier stack
                Imply()
            end for
        end if
    end if

Command: tpg -rtp v0 fault_coverage_number -jf v0 ATPG(PODEM or D ALG) tp_output.txt

Results: v0 with -jf / without -jf

D Algorithm

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.036/0.04	8/8	100
c2	0.079/0.08	13/12	88.6
c880	51.33/53.1	165/168	100

PODEM

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.039/0.046	8/9	100
c2	0.070/0.078	12/11	88.6
c880	47.75/50.2	168/168	100

Analysis: The -jf heuristic has little effect on runtime for small circuits like c17 and c2 but noticeably reduces runtime for larger circuits like c880 in both the D-algorithm and PODEM. This shows that the heuristic works well for complex circuits by focusing on inputs that are easier to control.
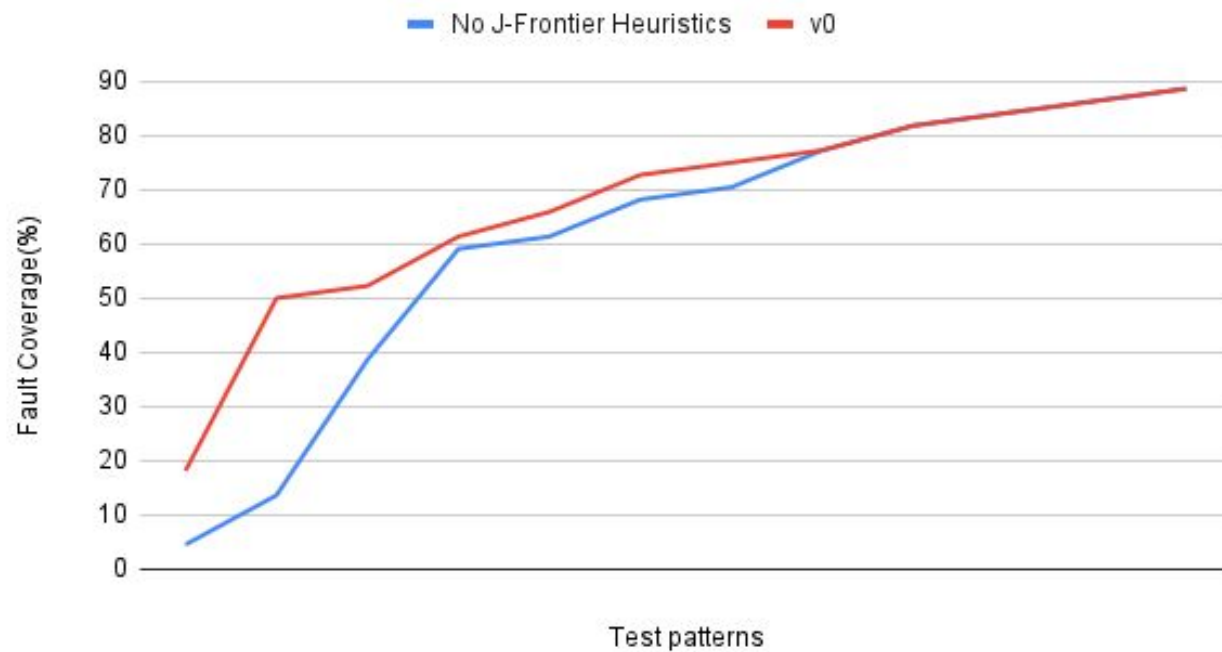
c17 D-Algorithm

c17 PODEM

Legend: — No J-Frontier Heuristics — v0

X-axis: Test patterns
Y-axis: Fault Coverage(%)

c2 D-Algorithm

No J-Frontier Heuristics — v0

c2 PODEM
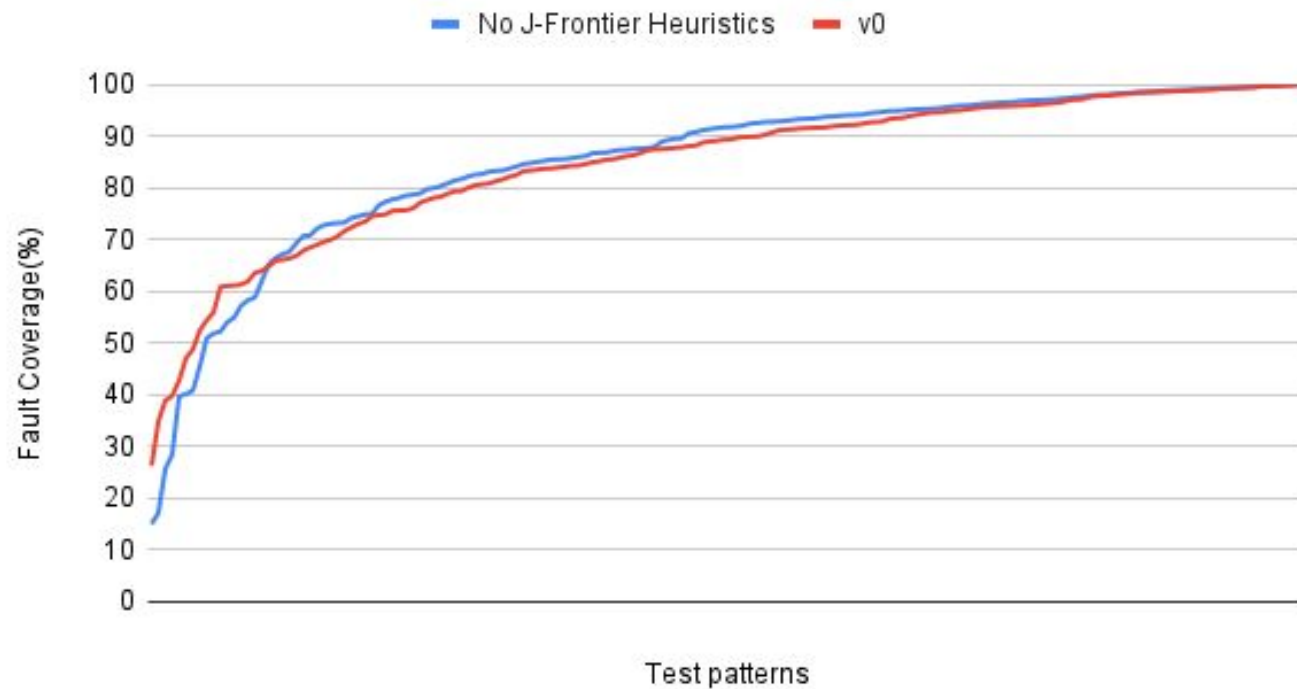
No J-Frontier Heuristics — **v0**

c880 D- Algorithm

No J-Frontier Heuristics   v0
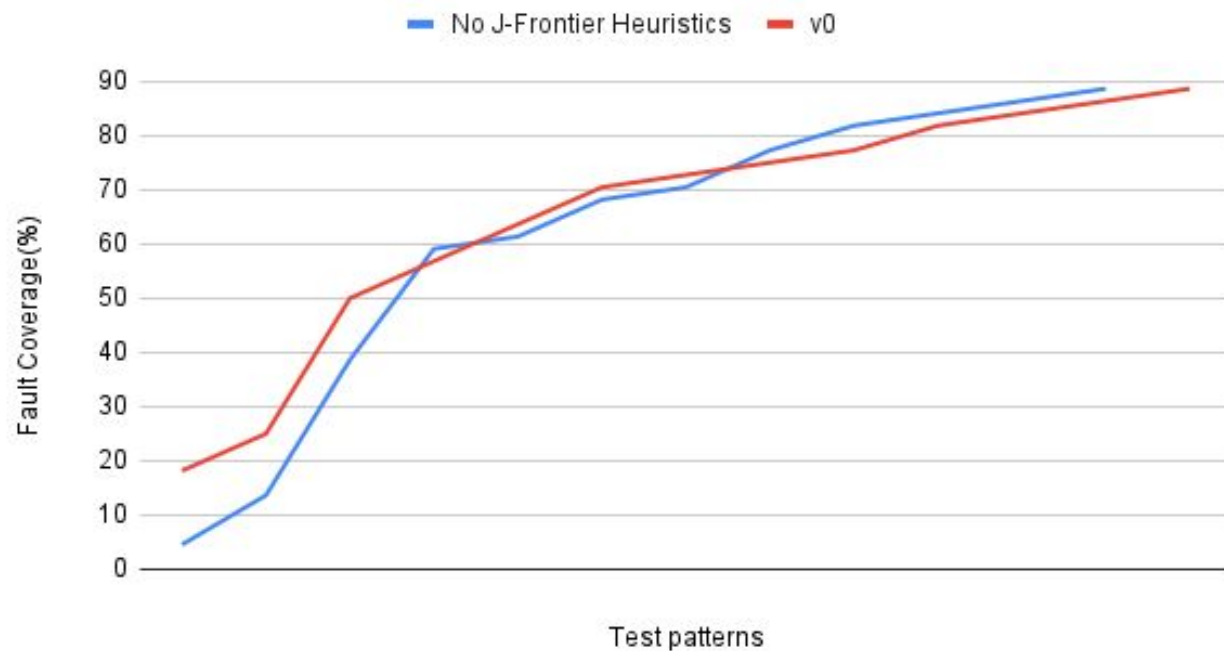
c880 PODEM

Intuition: We choose the fault with the lowest level because faults closer to the primary inputs have shorter backtracing paths, making it easier and faster to generate test patterns

Design:

Algorithm 1 Fault Order -fo v0
---
if Fault List is not empty **then**
   if -fo v0 **then**
      for Fault List sorted by level from small to large **do**
         node = top fault from fault list with the lowest level
         remove the fault from fault list
      end for
   end if
end if

Command: tpg -rtp v0 fault_coverage_number -fo v0 ATPG(PODEM or D ALG) tp_output.txt

Results: v0 with -fo / without -fo

D Algorithm

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.046/0.04	10/8	100
c2	0.065/0.08	11/12	88.6
c880	39.380/53.1	154/168	100

PODEM

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c17	0.046/0.046	10/9	100
c2	0.064/0.078	11/11	88.6
c880	37.94/50.2	153/168	100

Analysis: The fault order heuristic significantly reduces runtime, particularly for larger circuits, while maintaining fault coverage and comparable test volumes.
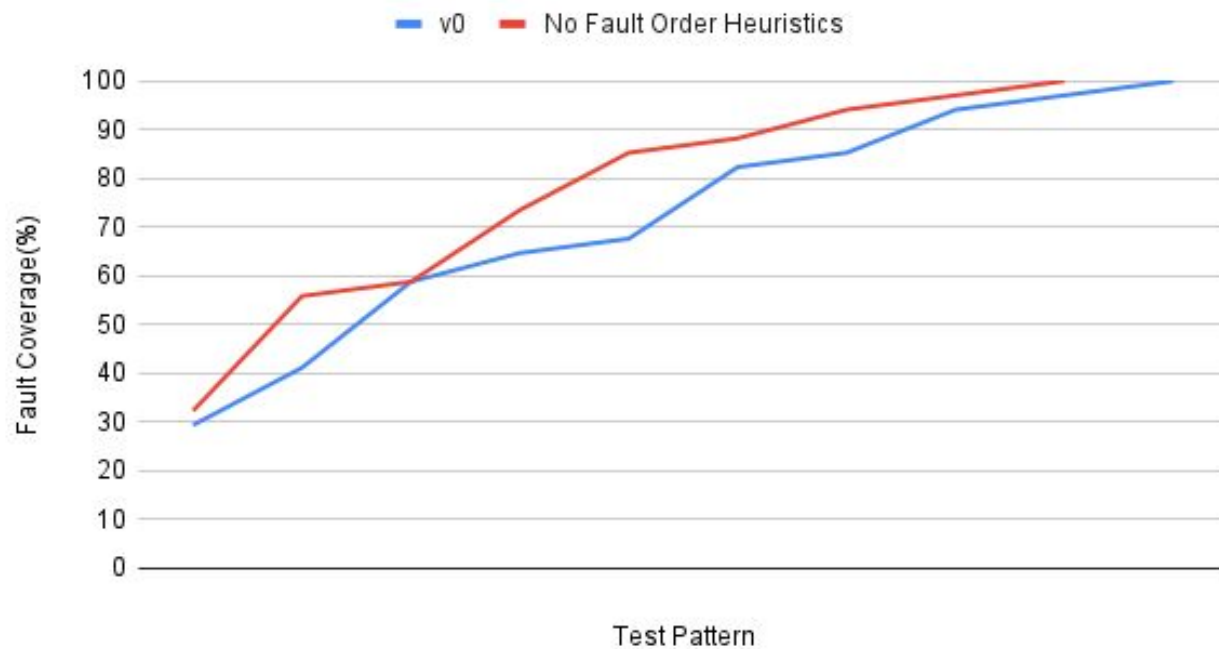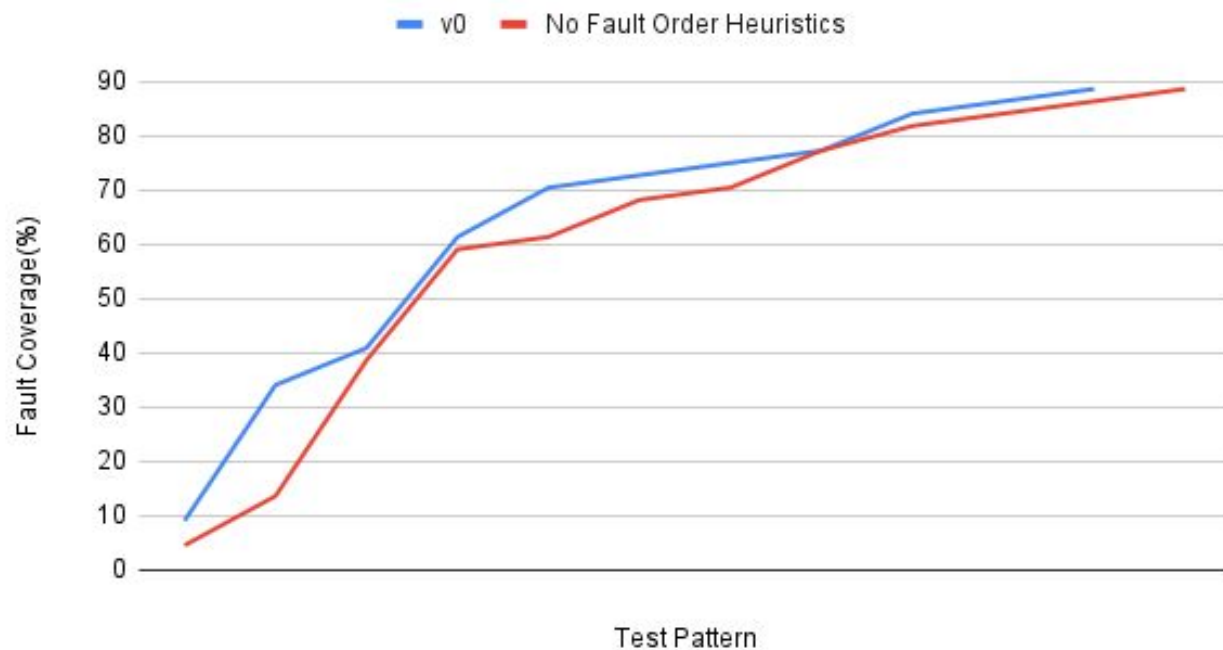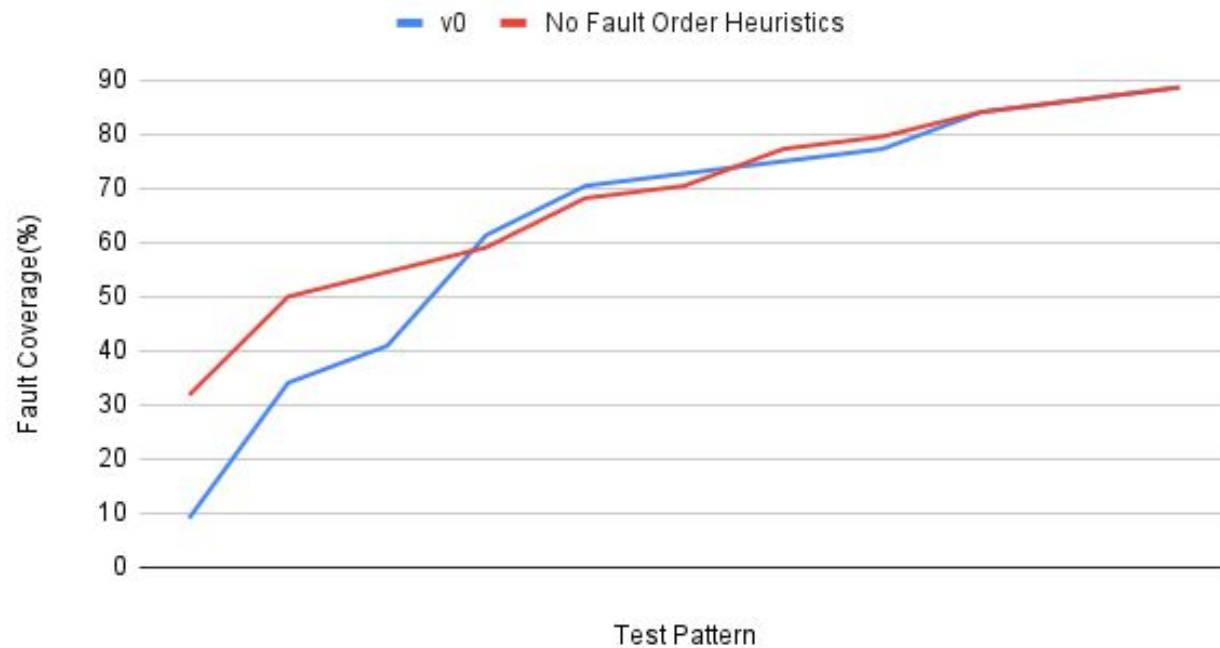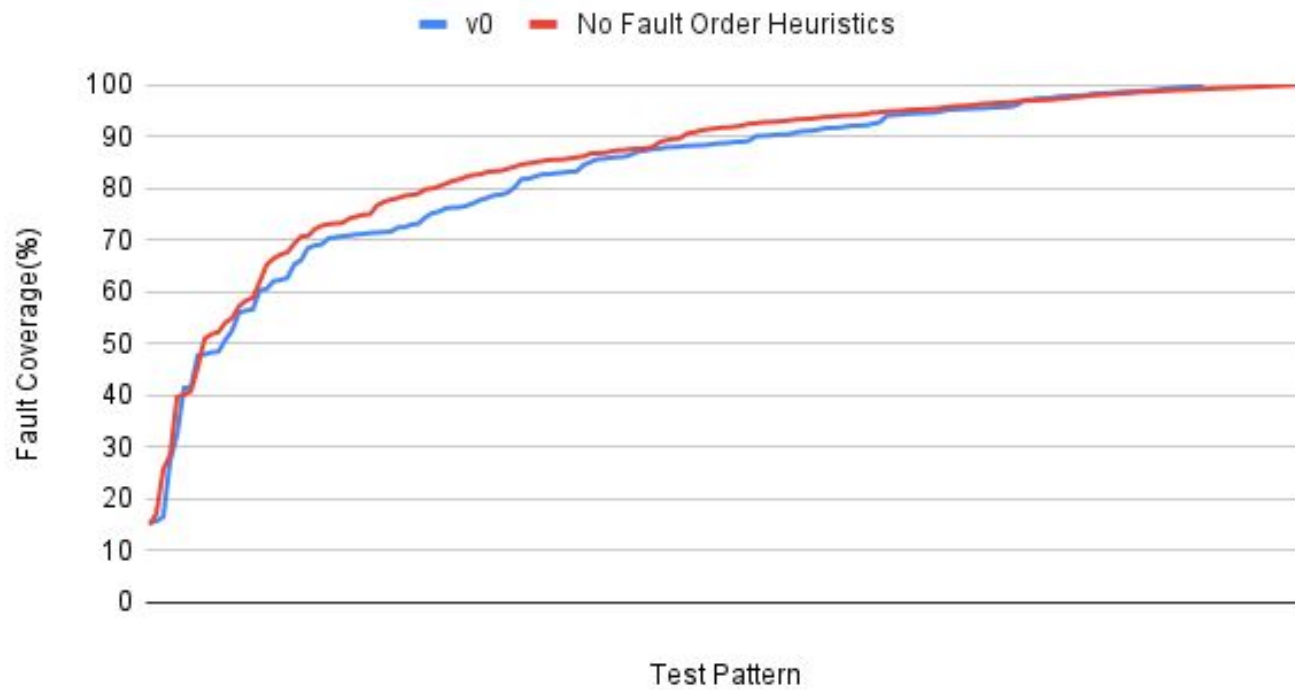
c17 D-Algorithm
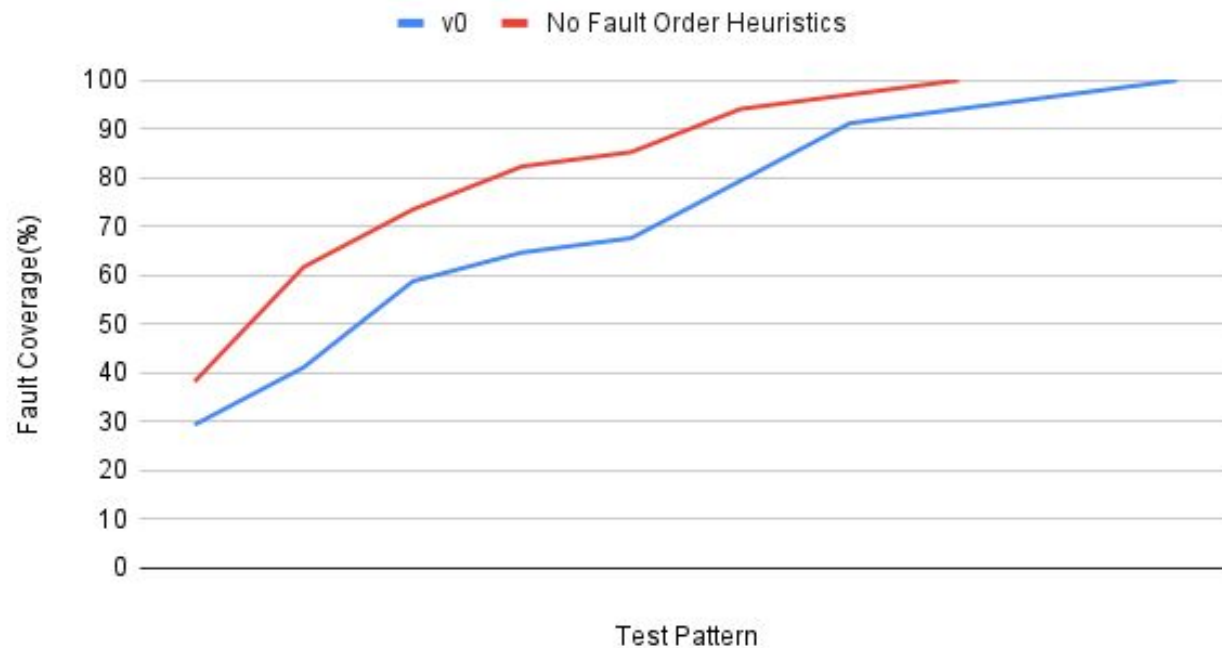
c17 PODEM

c2 D-Alorithm

c2 PODEM

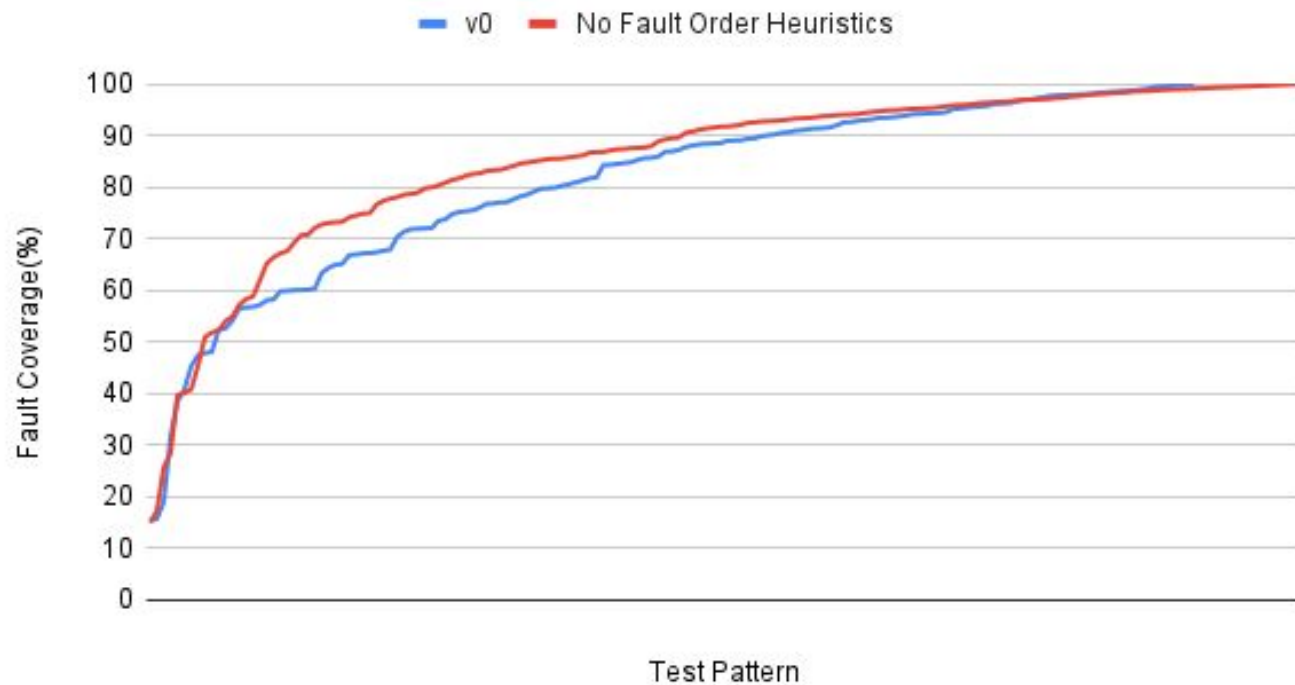c880 D-Algorithm

c880 PODEM

Description: Now we have our heuristics results. As we mentioned before, we want find the combination of heuristics to find which combinations has minimum time to reach the max fault coverage.

Intuition: we choose c880, rtp v0, fo, and PODEM, -df ll to test whether this combination can reduce more runtime or not.

**Command:** tpg -rtp v0 FC PODEM -fo v0 -df ll tp_output_fname

Results:

PODEM

	Runtime (s)	Test Volume (#tp)	Fault Coverage (%)
c880	37.9	144	100

Analysis:
 We can see that the runtime doesn't improve a lot compared with c880 with -fo v0. One of reasons I can know is that when we have command -df ll and -fo, we need to run levelizer fist. The levelizer takes some time to generate level for c880. Hence, this combination doesn't reduce the runtime a lot.

c880 PODEM Multi Heuristics

TEMPLATE
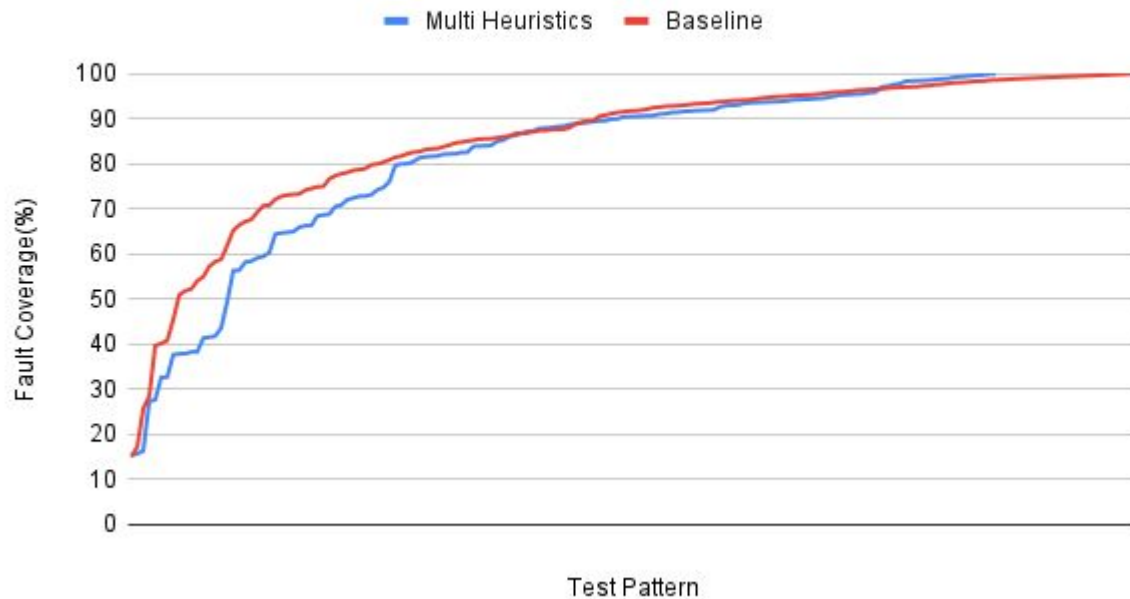
Intuition: As we observed in the results of rtpg-v0 that ….. was a problem, we also observed that this was a more serious issue in netlists with ….. characteristics. Our intuition is that if we utilize the netlist information, such as gate number, or ….., or ….. to determine hyper-parameter ….., it will address the ….., and we expect the ….. to be improved.

Design: It must be readable and clean. Your options are:
- **Pseudocode:** Tex (e.g. Overleaf), Markdown, or some online platform.
- **Flowchart:** It should follow standards. You can use draw.io or other online platforms.
- **Text editors:** Using a plain text with random indentation and style is definitely <u>NOT accepted</u>, However, it is OK to use <u>text editors only if required modifications</u> of font, style, indentation, etc. are made and the format is consistent through your pseudocodes.

Pseudocode/Flowchart do not fit?
- Use links!
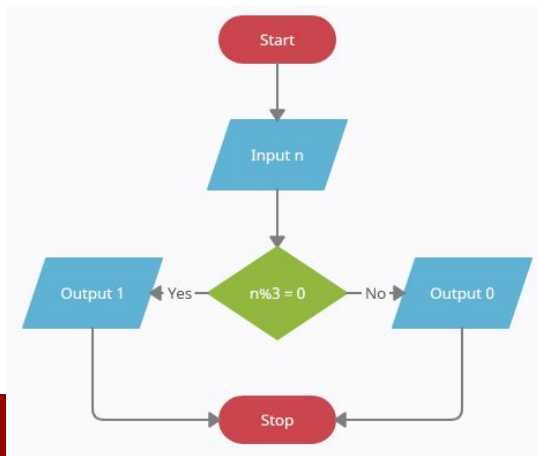
Algorithm 2: RTPG v4: Selecting the best random test pattern

```
...
while not done with RTPG do
    FL: list of currently remaining faults
    ...
    newTPs ← empty lists
    detFaults ← empty lists
    for i ← 1 to Q do
        newTPs[i] ← RTPG
        detFaults[i] ← PFS(FL, newTPs[i])    # do not update FL
    best-idx ← argmax(detFaults)
    best-tp ← newTPs[best-idx]
    # add best-tp to TPs, run PFS, update FL
    # maybe calculating ΔFC to decide termination?
    ...
return TPs
```

TEMPLATE

Results: The only acceptable result is what includes the large netlists.

- Tables: Create a Google Sheet, shared with your team and course staff, copy your tables.
- Figures: For some cases like RTPG, TPFC diagram can be very helpful.
- Make sure to report: fault coverage (FC), test volume (TV), and runtime (RT)
- Most probably you should report: improvements made to a previous or baseline heuristic

RTPG v4	c532	c588	c990	c1433	c2901	c3200	c6123	c9123
FC (%)	100	98.08	100	100	100	97.24	99.8	100
RT (s)	3.42	14.34	9.75	6.28	5.65	14.54	7.08	9.7
TV (#)	49	62	46	88	54	121	96	138
Δ RT (%)	-22.1	-4.3	2.5	26.6	37.2	48.3	69.8	143
Δ TP (%)	12.5	7.6	10.6	8.9	5.4	-6.8	2.3	-8.6

Table caption is always a must: *"RTPG v4 results compared with RTPG v3"*

$$\Delta\Phi = \frac{\Phi_{new} - \Phi_{baseline}}{\Phi_{baseline}} \times 100$$

TEMPLATE
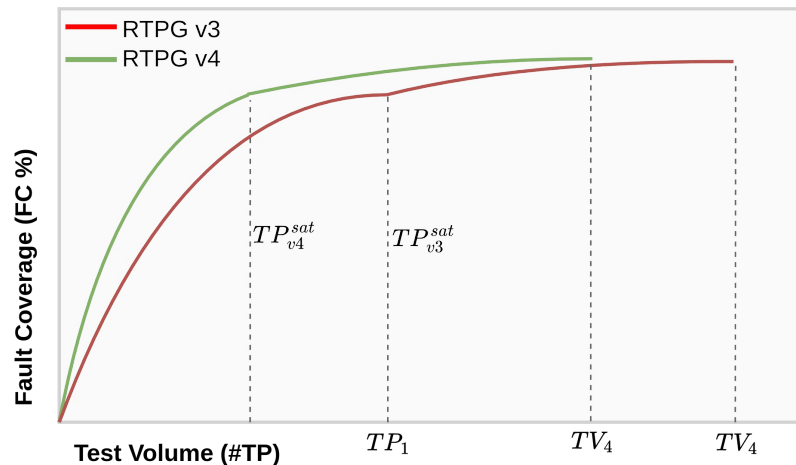
Command:

```
$ ./simulator -alg DALG -rtpg 32 64 alpha -flagA true 1
```

Analysis: As we expected, the blah improvement is very high for large circuits. The reason that the blah is decreased in some cases is because they are smaller circuits, and in the case of cXXXX as we saw in heuristic blah we could expect that the random approaches do not usually work well. We can see the difference of improvement when hyperparameter blah is changed from this to that. This observation and results gives us two suggestions on how to improve this further: how could we avoid blah in netlist blah, and how to detect blah in situation blah.