VLSI Testing Final Project Report

CIS 4930.009

Benjamin Gipalo – U16175094

Logan White – U64654965

Project Overview

Fittingly – the final project for our VLSI testing class requires us to create an interactive python program that generates vector tests to expose the unique faults that exist on a given circuit. As members of the undergraduate level course CIS 4930, we were tasked with implementing six menu items into our program:

- [0] Read in the Netlist => Parse input in terms of gates and connections to make it readable by our program.

- [1] Perform Fault Collapsing => Produce a list of fault classes that can exist on the circuit, minimized as redundant faults are condensed.

- [2] List Fault Classes => Print the unique fault classes identified by menu item [1], with one line per unique fault.

- [3] Simulate => Accepts an input vector alongside a set of faults and produces the outputs. If any faults are propagated to a primary output, it will be specified in the output.

- [4] Generate Tests (D-Algorithm) => Follows the D-Algorithm covered extensively in class to produce a list of test vectors for each fault class identified by menu item [1].

- [7] Exit => Terminates the Python program.

The referential patterns between these menu items reveal a clear program flow. Users start by using [0] to read a circuit as input. Then, they use [1] to get the program to produce a minimized fault list. Then, users can freely use [2], [3], and [4] to test their circuit or generate tests to expose faults.

Once all six menu items have been implemented, we can test our program by running it on the benchmarks provided on Canvas.

Code Structure – Menu Class and Module Implementation

Our code is structured as a python module called “ATG_SSF”, with its entry point in __main__.py. Structuring our project in this way enabled us to more easily set up dialog for optional graphing capabilities, as well as make the CLI prettier.

It is important to note that attributes of “Menu” are used to track the state of our understanding of the circuit. Exactly one Menu is instantiated per usage of the program inside __main__py. The Menu class has properties like “gates” “fault_list” and “file_lines”.

Whenever a user launches the module, they specify a single positional argument for the file that represents the circuit to be tested. Then, as the very first menu items are printed to the user terminal, each line in the input file is set on Menu’s “file_lines”, and all Menu’s other attributes are set to Null.

Once this happens, the program is ready for the user to select option [0], where it will read from Menu’s “file_lines” to populate Menu’s “gates” and “graph” attributes.

These “gates” and “graph” attributes are our key to representing simple circuits in code.

Code Structure – Gate Representation

In our program, gates are identified by the name of their output wire. In addition to this, we also track the type (e.g. AND, OR, NAND), a list of inputs as strings, and level for each gate. For example, if sole primary inputs 1gat and 2gat are connected to an OR gate, our program would represent the OR gate as follows inside Menu's "gates":

"3gat": {

  "type": "OR",

  "inputs": ["1gat", "2gat"],

  "level": 1

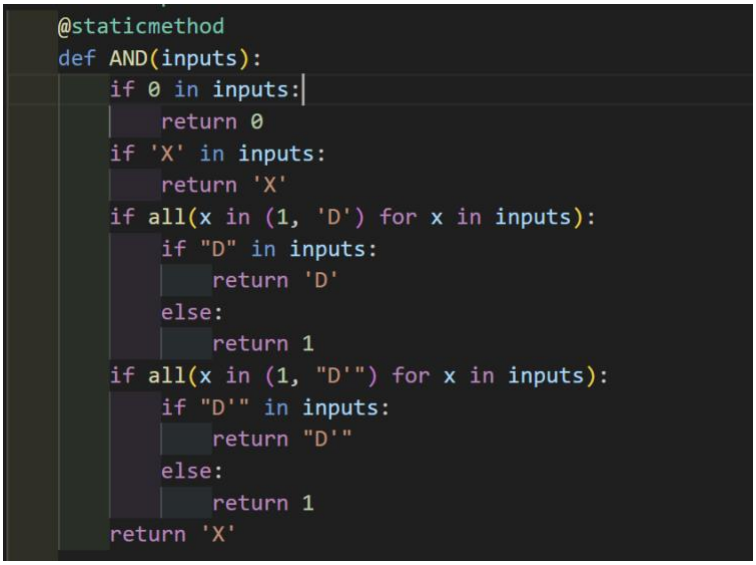
}

Since fault collapsing concerns itself with PIs and POs only, our program identifies each during the process of populating Menu's "gates". Firstly, all PIs will have their "type" set to "PI", and their "level" set to 0. Then, all POs will have their "level" manually set to the maximum level naturally occurring in the circuit. With both relationships identified, we can reference our PIs and POs at any time by checking the type and level of a single pass-through Menu's "gates".

This covers how our program represents gates in a circuit, but we still must cover how our program handles gate logic. A major part of this course was understanding simple gates (AND, OR, NAND, NOR, NOT) in terms of their control and inverse values.

By understanding the behavior of each gate's control and inverse values, we can implement static functions for each gate that use a series of conditionals. Each static function can return one of the five values covered in class: 0, 1, X, D, or ~D.

For example, here is our static function for the AND gate:



```
@staticmethod
def AND(inputs):
    if 0 in inputs:
        return 0
    if 'X' in inputs:
        return 'X'
    if all(x in (1, 'D') for x in inputs):
        if "D" in inputs:
            return 'D'
        else:
            return 1
    if all(x in (1, "D'") for x in inputs):
        if "D'" in inputs:
            return "D'"
        else:
            return 1
    return 'X'
```

- Since the AND gate control value is 0, we return 0 XOR i = 0 if it is present in the input.
- If any input is unknown, our output is also unknown.
- Propagate input fault to output if present.
- In all other cases, output is unknown.

With chosen structures to represent individual gates, as well as processing gate logic, we can move onto the implementation of individual line items.

Code Structure – Fault Collapsing

As covered in class, when performing Fault Collapsing, we start with the subset of faults that can exist on Primary Inputs and Primary Outputs.

Then, the code iterates over each non-PI gate in Menu's "gates", sourcing the control (c) and inverse (i) values for that gate type.

The program starts by checking the gate for functionally redundant faults:

- An input stuck-at-c XOR i fault is functionally equivalent to the output stuck-at-c XOR i. It can be removed if there is no input fanout.

Then, the program checks the gate for any dominating faults:

- An input stuck-at-~c XOR i is dominated by an output stuck-at-~c XOR i. It can be removed if there is no fanout.

After checking for both redundancy and dominance, the program enforces that Menu's "fault_list" is minimized, needed for menu items [2], [3], and [4].

Code Structure – Simulation

The first step of implementing line item [3] is to prompt the user for faults to activate, as well as an input vector to test. Our program accomplishes this in a two district CLI phases:

In the fault selection phase, the program prints all the unique faults identified from line item [1] with an index for each. Then, users can input the indices of the faults they wish to be activated in a space separated list, stored in “chosen_faults”.

In the input vector phase, the program accepts a 0/1 for each PI, space separated. Each input will be stored in “chosen_inps”.

After this initial CLI is run, the simulation process initializes itself by setting three things on the gate matrix:

- Each PI is set to the value specified in the user provided input vector.
- Each output has its value set to “X”, representing that it is unknown.
- If any enabled faults are on primary inputs, override the user provided input (“D” or “~D” if applicable).

After this initial setup, the program can begin its main loop, which concludes when every gate output is known, meaning no gate output is “X”. As we visit gates in this loop, there are three possible outcomes:

- If the gate is a PI, continue as there is nothing to process.
- If all gate inputs are known, invoke its static function to set the output. Check activated faults to override gate output if applicable.
- If at least one gate input is unknown, process the corresponding gate with unknown output first.

By structuring our main loop in this way, processing the values of unknown gates as they occur, makes this an implementation of event driven simulation.

After all gate outputs are known (1, 0, D, or ~D), the simulation results are output:

If faults were injected, the values are each gate are compared in the “healthy” and “faulty” versions of the circuit. This helps highlight how the selected faults impact the circuit, as well as the path they take if they are propagated.

Finally, the value at each gate is printed, with any applicable faults labeled

Code Structure – D-Algorithm Test Generation

The goal of line item [4] is to generate an unopinionated test vector for each unique fault discovered in line item [1] that propagates that fault to a primary output.

By following the D-Algorithm process of propagating using D-Frontiers and justifying using J-Frontiers, we can propagate an injected fault to a primary output:

- D-Frontiers have at least one faulty input but still has an “X” output. We need to ensure that the faulty “D” or “~D” is propagated to that “X” output.

- J-Frontiers have a defined output, but at least one unknown input. We need to justify the inputs to ensure the desired output is achieved.

- We use the first implication after injecting the fault to create the first D-Frontier. Setting a s-a-0 line to 1 creates a faulty “D” value, and setting a s-a-1 line to 0 creates a faulty “~D” value.

- “Solving” a D-Frontier recursively creates new D and J-Frontiers:

    - Defining an input on a D-Frontier either creates a new J-Frontier or implicates a PI.

    - Propagating the fault to the output either creates a new D-Frontier or detects that fault at a PO.

- Exploring the tree of possible D/J-Frontiers will lead to two possible outcomes:

    - All D/J-Frontiers are resolved, and a faulty “D” or “~D” has been propagated to a PO. The current state of PIs can be returned, with “X” inputs being treated as don’t cares.

    - All possibilities lead to a contradiction, and the fault is undetectable.

To aid with the process outlined above, we will implement helper functions like “get_D_frontier”, “get_J_frontier”, and “inject_fault”. These functions traverse the current state of our circuit to identify gates or augment the circuit state in an intentional way.

For each unique fault, the first step in generating a test vector is to activate the fault. If it is a stuck-at-0 fault, this means forcing a gate output to “D”. If it is a stuck-at-1 fault, this means forcing a gate output to “~D”. Forcing that gate output is our first implication, and the resulting “D” or “~D” forms our first D-Frontier.

Since our D-Algorithm function is defined recursively, the first step is to check that all the logic for the current circuit is internally consistent. If there are any inconsistencies, “None” is returned and the process backtracks to the previous step.

The next step is to check if the fault (“D” or “~D”) has been propagated to a primary output. If it has, we can return the current PI as the working test vector.

If not, the next step is to propagate the existing fault closer to a primary output. As described above, this process involves turning an existing D-Frontier into a J-Frontier (located at the non-faulty input) and a new D-Frontier (located at the existing D-Frontier's output).

Then, the next step is to set the unknown inputs on the new J-Frontier to ensure that the existing D-Frontier can propagate its fault.

Since this process is recursive, each time we justify a new value, we enter a deeper layer of recursion. Each time we return “None”, we terminate a layer of recursion and return to the previous layer.

On successful propagation, we can use the current state of PIs with “X” as don’t cares as a test vector to detect the injected fault.

On failure, the injected fault cannot be propagated to a primary output, and is therefore undetectable.

At the highest level, we iterate over the minimized fault_list. For each fault, we inject (and activate) that fault on the circuit before running the recursive D-Algorithm to either generate a test vector or determine that the fault is undetectable. After doing this for each fault, we can cleanly output each fault class alongside the test that can detect it.

This output is structured as a simple table, with rows and columns representing unique faults and primary input values, respectively. In this table, the first justification that activates the injected fault will be highlighted in red.

Compile Instructions

```
git clone https://github.com/gipalob/VLSI-Testing-Final-Proj.git
pip install -r VLSI-Testing-Final-Proj/requirements.txt
```

Run Instructions

View Run Instructions

```
python -m ATG_SSF
```

Start Analysis

```
python -m ATG_SSF <path_to_circuit_file>
```

Benchmark Results

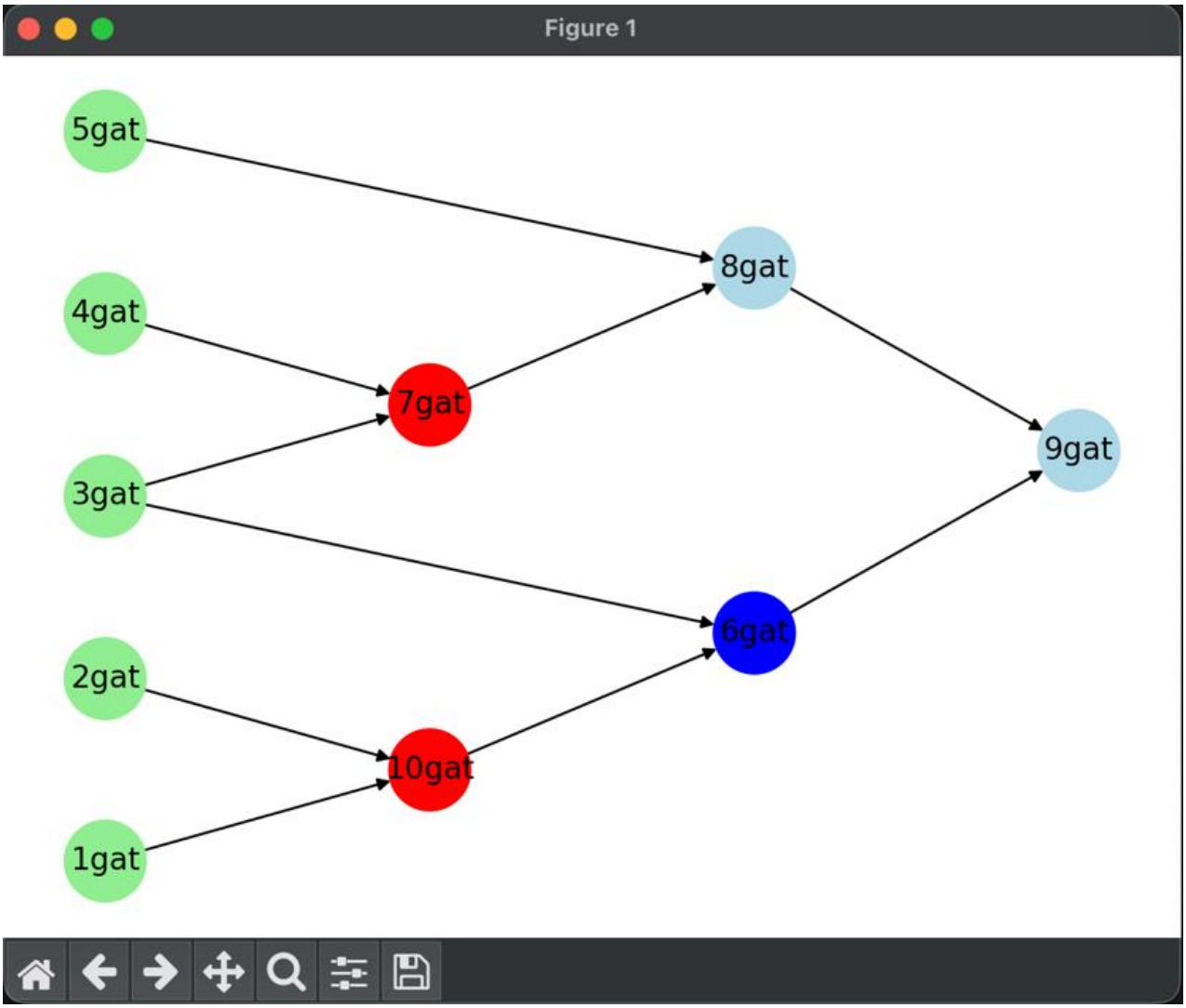
For viewing circuit visualizations, this color table will be useful:

Color	Gate Type
Light Green	PI
Light Blue	OR
Blue	NOR
Red	AND
Coral	NAND
Pink	XOR

t4_21.ckt

```
Would you like to enable additional features? ('Y' / 'N' / 'info'): y
Additional features enabled.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[5]: Generate tests (PODEM)
[6]: Generate tests (Boolean Satisfaibility)
[7]: Exit
Selection: 0
        Netlist processed successfully.
        Would you like to view the circuit's graph visualization? ('Y' / 'N'): y

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[5]: Generate tests (PODEM)
[6]: Generate tests (Boolean Satisfaibility)
[7]: Exit
Selection: 1
        Fault collapsing completed successfully.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[5]: Generate tests (PODEM)
[6]: Generate tests (Boolean Satisfaibility)
[7]: Exit
Selection: 4
        Generating tests using D-Algorithm.........
        All possible vectors generated!

Test Vectors:('DC' indicates a don't care input)
There is a risk that not all DCs will be caught- however, the DCs that are caught are true DCs.

Fault   | PI Assignments
        | 1gat  | 2gat  | 3gat  | 4gat  | 5gat
-----------------------------------------------
1gat s-a-0 | 1  | 1     | 0     | DC    | 0
2gat s-a-1 | 1  | 0     | 0     | DC    | 0
3gat s-a-0 | 0  | DC    | 1     | 1     | 0
3gat s-a-1 | 0  | DC    | 0     | 1     | 0
4gat s-a-1 | 0  | DC    | 1     | 0     | 0
5gat s-a-1 | 0  | DC    | 0     | DC    | 0
6gat s-a-1 | 0  | DC    | 1     | 0     | 0
9gat s-a-0 | 1  | 1     | 1     | 1     | 1
9gat s-a-1 | 0  | DC    | 0     | DC    | 0
```

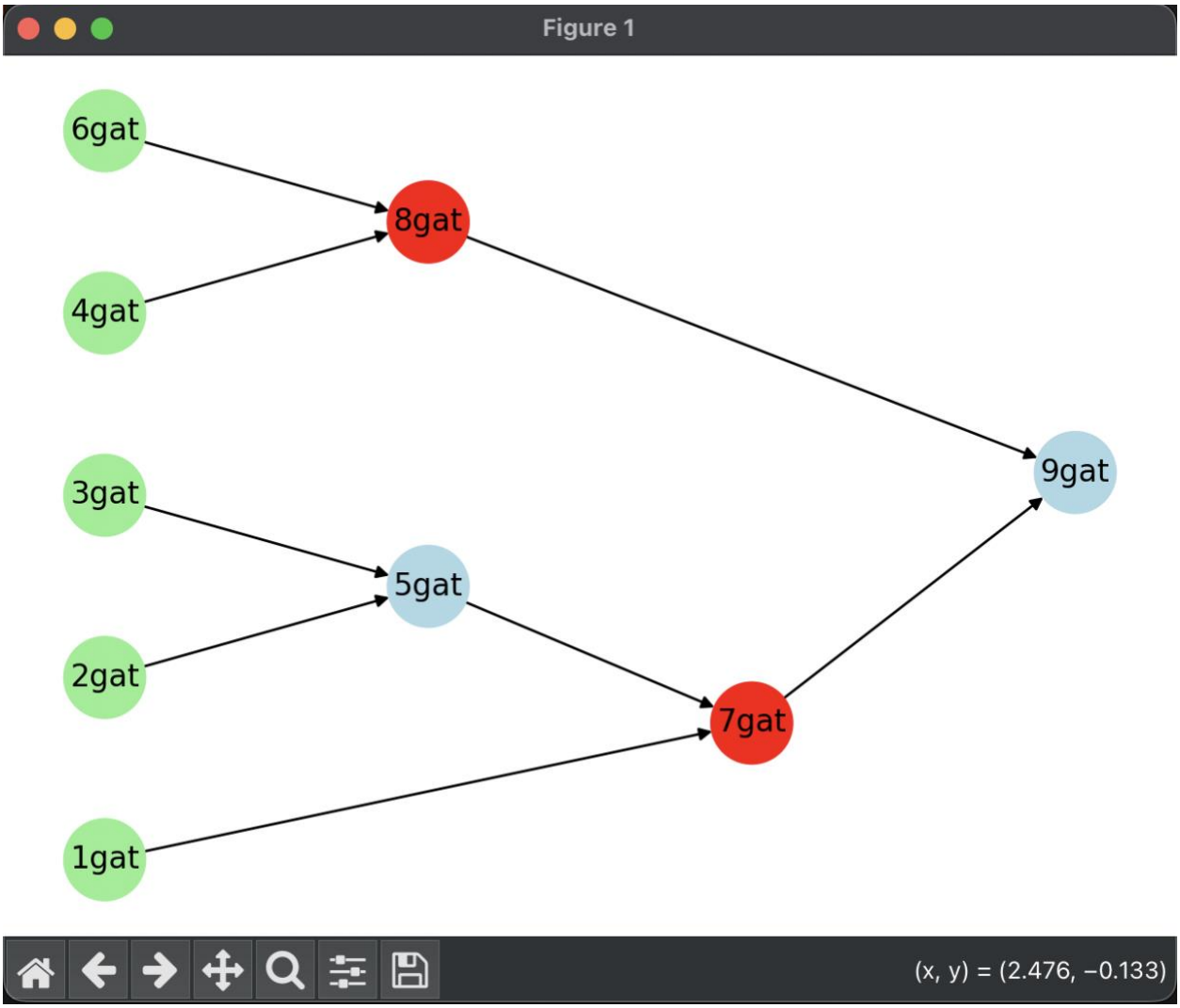
Circuit visualization, after running [0] Read the input net-list



t4_3.ckt

```
Would you like to enable additional features? ('Y' / 'N' / 'info'): y
Additional features enabled.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[5]: Generate tests (PODEM)
[6]: Generate tests (Boolean Satisfaibility)
[7]: Exit
Selection: 0
        Netlist processed successfully.
        Would you like to view the circuit's graph visualization? ('Y' / 'N'): y

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[5]: Generate tests (PODEM)
[6]: Generate tests (Boolean Satisfaibility)
[7]: Exit
Selection: 1
        Fault collapsing completed successfully.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[5]: Generate tests (PODEM)
[6]: Generate tests (Boolean Satisfaibility)
[7]: Exit
Selection: 4
        Generating tests using D-Algorithm........
        All possible vectors generated!

Test Vectors:('DC' indicates a don't care input)
There is a risk that not all DCs will be caught- however, the DCs that are caught are true DCs.

Fault   | PI Assignments
        | 1gat  | 2gat  | 3gat  | 4gat  | 6gat
-----------------------------------------------
1gat s-a-0 | 1  | 1     | DC    | DC    | 0
2gat s-a-1 | 1  | 0     | 0     | DC    | 0
3gat s-a-0 | 1  | 0     | 1     | DC    | 0
4gat s-a-1 | 0  | 0     | 0     | 0     | 1
6gat s-a-0 | 0  | 0     | 0     | 1     | 1
7gat s-a-1 | 0  | 0     | 0     | DC    | 0
9gat s-a-0 | 1  | 1     | DC    | 1     | 1
9gat s-a-1 | 0  | 0     | 0     | DC    | 0
```

Circuit visualization, after running [0] Read the input net-list



t5_10.ckt

```
(base) bgipalo@Bens-MacBook-Pro VLSI-Testing-Final-Proj %  python -m ATG_SSF ./benchmarks/t5_10.ckt
Would you like to enable additional features? ('Y' / 'N' / 'info'): y
Additional features enabled.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 0
        Netlist processed successfully.
        Would you like to view the circuit's graph visualization? ('Y' / 'N'): n

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 1
        Fault collapsing completed successfully.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 4
        Generating tests using D-Algorithm.........
        All possible vectors generated!

Test Vectors:('DC' indicates a don't care input)
There is a risk that not all DCs will be caught- however, the DCs that are caught are true DCs.

Fault   | PI Assignments
        | agat  | bgat  | cgat  | dgat  | egat
-----------------------------------------------
agat s-a-0 | 1  | 1     | 0     | DC    | 0
bgat s-a-1 | 1  | 0     | 0     | DC    | 0
cgat s-a-1 | 0  | DC    | 0     | 1     | 0
dgat s-a-1 | 1  | 1     | 1     | 0     | 0
egat s-a-0 | 1  | 1     | 0     | DC    | 1
fgat s-a-0 | 1  | 1     | 0     | DC    | 0
igat s-a-1 | 1  | 1     | 0     | DC    | 0
jgat s-a-1 | 1  | 1     | 1     | 0     | 0
mgat s-a-0 | 1  | 1     | 1     | 1     | 1
mgat s-a-1 | 1  | 1     | 1     | 0     | 0
```

Circuit visualization, after running [0] Read the input net-list

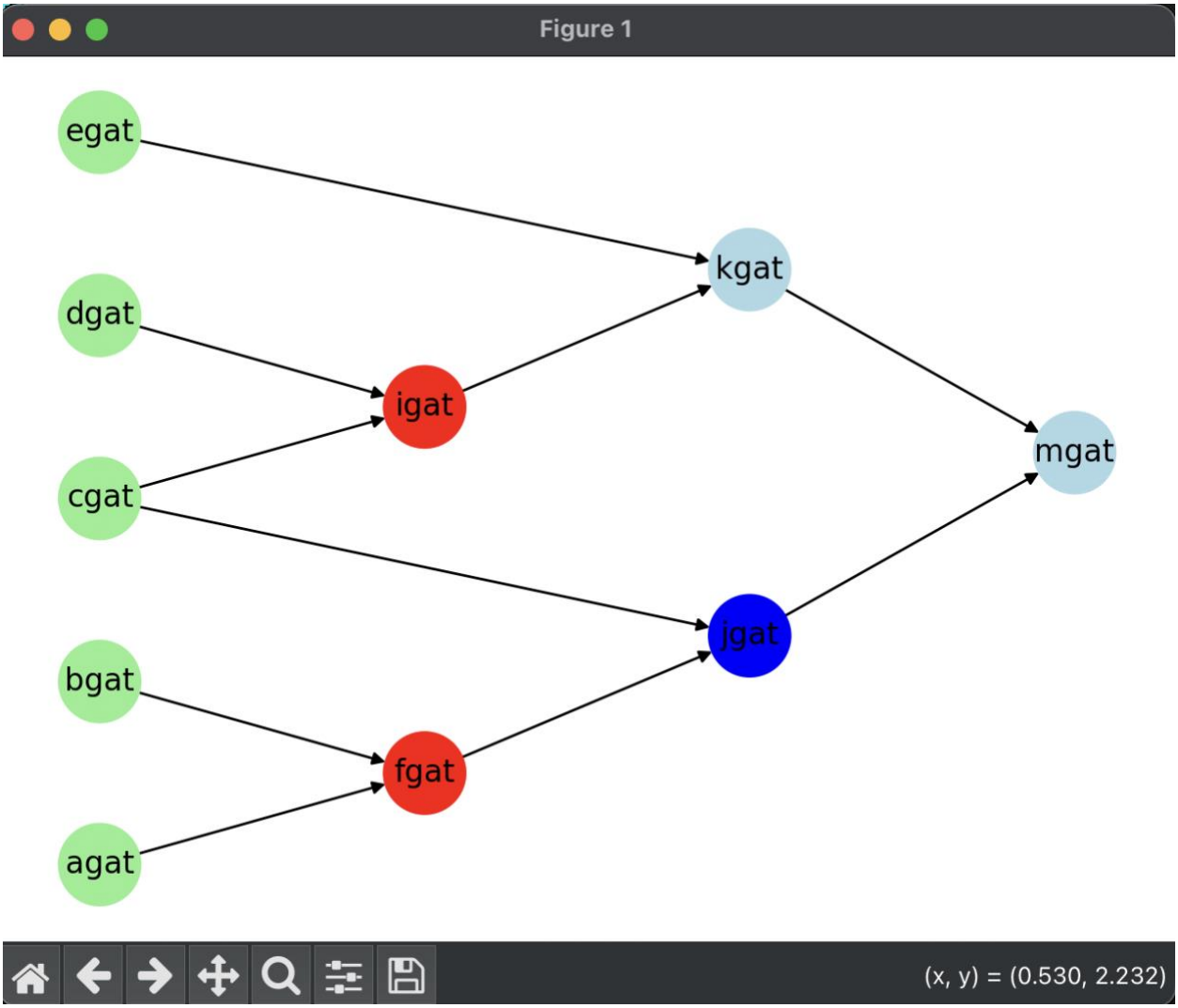
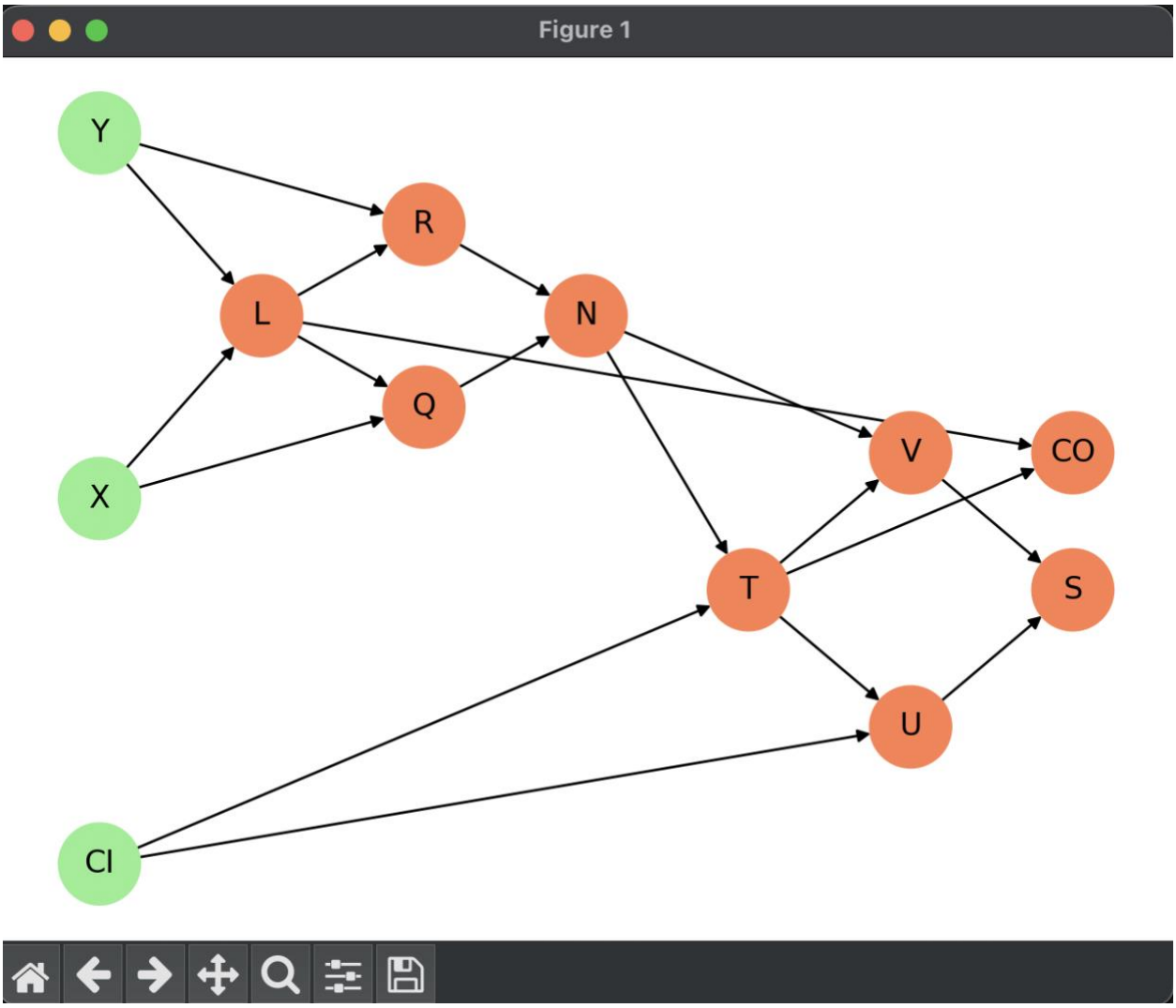


Figure 1
egat
dgat
cgat
bgat
agat
igat
fgat
kgat
jgat
mgat
(x, y) = (0.530, 2.232)

t5_26a.ckt

```
(base) bgipalo@Bens-MacBook-Pro VLSI-Testing-Final-Proj %  python -m ATG_SSF ./benchmarks/t5_26a.ckt
Would you like to enable additional features? ('Y' / 'N' / 'info'): n
Additional features disabled.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 0
        Netlist processed successfully.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 1
        Fault collapsing completed successfully.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 4
        Generating tests using D-Algorithm

No test found for CI s-a-0


No test found for CI s-a-1


No test found for X s-a-0


No test found for X s-a-1


No test found for Q s-a-1


No test found for N s-a-0


No test found for U s-a-1


No test found for CO s-a-1


No test found for S s-a-0


No test found for S s-a-1

        All possible vectors generated!

Test Vectors:('DC' indicates a don't care input)
There is a risk that not all DCs will be caught- however, the DCs that are caught are true DCs.

Fault   | PI Assignments
        | CI    | X     | Y
-------------------------------------------------
Y s-a-0 | 1     | 1     | 1
L s-a-0 | 1     | 0     | 1
T s-a-0 | 0     | 1     | 1
CO s-a-0 | 1    | 0     | 0
```

Circuit visualization, after running [0] Read the input net-list



t6_24_v1.ckt

```
(base) bgipalo@Bens-MacBook-Pro VLSI-Testing-Final-Proj %  python -m ATG_SSF ./benchmarks/t6_24_v1.ckt
Would you like to enable additional features? ('Y' / 'N' / 'info'): n
Additional features disabled.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 0
        Netlist processed successfully.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 1
        Fault collapsing completed successfully.

Main Menu:
[0]: Read the input net-list
[1]: Perform fault collapsing
[2]: List fault classes
[3]: Simulate
[4]: Generate tests (D-Algorithm)
[7]: Exit
Selection: 4
        Generating tests using D-Algorithm

No test found for 4gat s-a-0


No test found for 5gat s-a-0


No test found for 6gat s-a-0


No test found for 10gat s-a-0


No test found for 12gat s-a-0


No test found for 14gat s-a-0


No test found for 19gat s-a-0


No test found for 16gat s-a-1

        All possible vectors generated!

Test Vectors:('DC' indicates a don't care input)
There is a risk that not all DCs will be caught- however, the DCs that are caught are true DCs.

Fault   | PI Assignments
        | 1gat  | 2gat  | 3gat  | 4gat  | 5gat  | 6gat
-------------------------------------------------------
1gat s-a-0 | 1  | 1     | 1     | 1     | 1     | 1
1gat s-a-1 | 0  | 1     | 1     | 1     | 1     | 1
2gat s-a-0 | 1  | 1     | 1     | 1     | 1     | 1
2gat s-a-1 | 1  | 0     | 1     | 1     | 1     | 1
3gat s-a-0 | 1  | 1     | 1     | 1     | 1     | 1
3gat s-a-1 | 1  | 1     | 0     | 1     | 1     | 1
17gat s-a-0 | 0 | 1     | 1     | 1     | 1     | 1
21gat s-a-1 | 0 | 0     | 0     | 1     | 1     | 0
16gat s-a-0 | 0 | 0     | 0     | 1     | 1     | 0
```

Circuit visualization, after running [0] Read the input net-list