Combinational D-Algorithm

L Lakshmanan, Ananya Sane, Jaishnav Yarramaneni, Eswara Rohan

Abstract

This report is for the Design for Testability course project, which required our team to design a software that is capable of taking a circuit in netlist format as an input (subject to limitations), and generating test vectors to test specific stuck-at faults by using the D-Algorithm. The software takes the circuit as input and simulates faults at all the edges, and finds test vectors (if they exist) for said faults. Our implementation of the D-Algorithm involves using functions specifically to find d-frontiers and j-frontiers, and perform justifications and backtrack to find the test pattern. The code is attached with the submission, and sample outputs and inputs are given in the report.

1 Introduction

Test pattern generation is one of the most important steps in testing the design in VLSI circuits. This is done with the help of an Automatic Test Pattern Generator (ATPG). The ATPG uses algorithms to generate test patterns and the D-algorithm is one such method of deterministic test pattern generation along with the PODEM and FAN algorithms. D-algorithm defines a construct called the D-algebra, which is used for the test pattern generation. This algorithm uses concepts like fault activation, propagation, justification, and backtracking to find the test vector that will enable us to detect the fault if the fault is testable.

2 Problem Statement

The main problem statement for the project is to construct a software that can take circuits as inputs and give us the corresponding output test vectors for possible faults in the circuit by using the D-Algorithm.

3 Working of the D-Algorithm

The D-Algorithm is implemented with the help of the D-Algebra, which consists of a 5-value logic given by 1, 0, D, D', and X. The steps followed in the algorithm are as follows.

- Fault Activation or creating a D-Frontier.
- Fault Effect propagation or driving D-Frontier towards the output.
- Find the suitable inputs justifying the Fault or Justifying J-Frontier.
- Backtracking if any conflict occurs at some node.

3.1 Create D-Frontier or Fault Activation

- A D-Frontier is a set of gates with D or D' at the inputs and X at the output.
- Given a fault we try to generate a Primitive D Cube which is used to specify minimum input conditions required at a gate to propagate D or D' to the output.
- Assign the values of Primitive D Cube to the gate inputs thereby generating a D-Frontier. This is the fault activation step and we get D or D' at the output of the gate after this.

3.2 Drive D-Frontier towards output or Fault Propagation

- After creating a D-Frontier we drive the D-Frontier towards output using Propagation D Cubes (PDCs).
- Propagation D Cube is the minimum gate input assignments required to propagate a D/D' from gate inputs to gate output.
- In case of fan-outs, choose a single path for propagation. If the fault effects cannot be propagated, backtrack to the last decision point and change choice.

3.3 Justify J-Frontier

- J-Frontier is a set of gates whose output value is assigned but the input values are not yet decided (X).
- Justification step involves assigning input values to the gates so that a consistent set of circuit input values that gives the required value at the fault site.

3.4 Backtracking at conflict

- Backtracking involves changing the choice at the last decision point if there is any inconsistency found with the assigned values.

3.5 Flow Chart of D-Algorithm

The flowchart can be seen for a better understanding and visualisation of the flow of the algorithm in Figure 1.

4 Algorithm Implementation

- The programming language used for this project is C++ with the help of STL library. The language was chosen because of its speed and high level implementation, both of which are benefits and aid our construction.
- The main data structures used in implementing this project include Graphs, Arrays, Pairs and Vectors.

A circuit is built and is given as an input in a specific format.

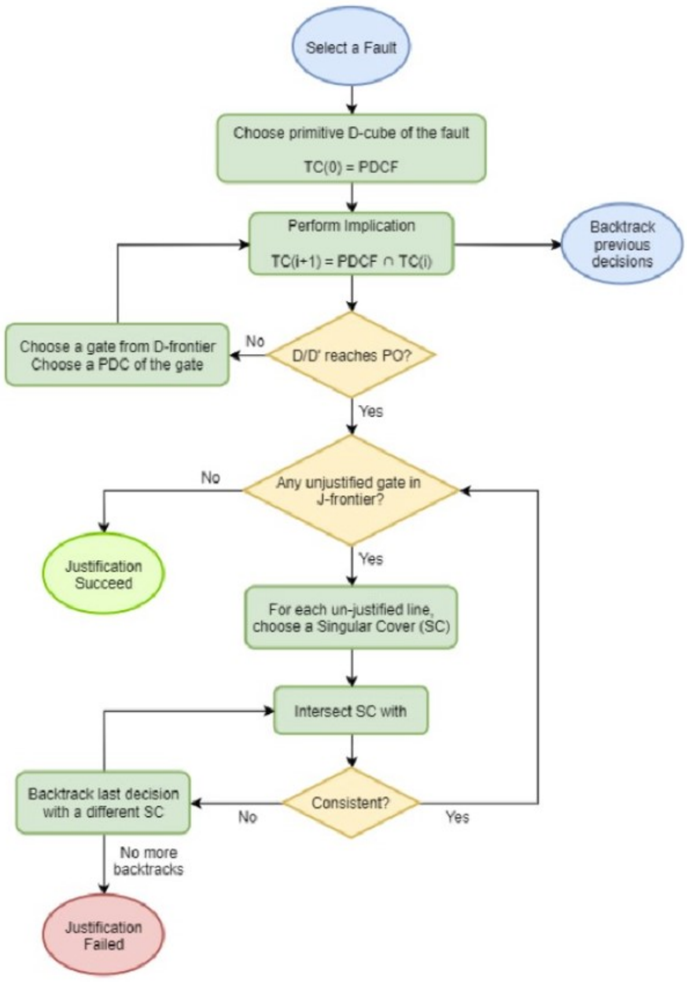
Select a Fault
Choose primitive D-cube of the fault
TC(0) = PDCF
Perform Implication
TC(i+1) = PDCF ∩ TC(i)
Backtrack previous decisions
Choose a gate from D-frontier
Choose a PDC of the gate
No
D/D' reaches PO?
Yes
No
Any unjustified gate in J-frontier?
Justification Succeed
Yes
For each un-justified line, choose a Singular Cover (SC)
Intersect SC with
Backtrack last decision with a different SC
No
Consistent?
Yes
No more backtracks
Justification Failed

Figure 1: Flowchart describing general flow of D-Algorithm

4.1 Combinational circuit implementation

- The combinational circuit is represented with the help of graphs.
- Each node of the graph represents either a gate, primary inputs, primary output or a branchout.
- The gate, primary inputs, primary outputs and the branches are identified using a specific number which determines their functionality of the type of gate or the stuck-at-fault or fault free. Edges of the graph are wires, and are indexed starting from 50 to prevent conflict in naming.

The circuit is then simulated with all possible single-stuck-at faults and the test pattern is generated for each of them if possible.

4.2 Generating the Test Vector

Generation of the test vector involves the following steps and functions:

- Initialise test cube to all x's.
- **addWire():** Basic function to add wires as edges in the graph.
- **printCirc():** Function to print the adjacency list that constructs the circuit.
- **getwire_sur():** Takes wire as input and finds the gates at the input and output of the given wire, as well as the other wires that are connected to the above gates.

- **getbranch_sur():** Takes node (gate) as input and finds the wires and gates connected to the fanout if the gate is a fanout node.
- **dFrontier():** Adds gates with x at the output to the list of D Frontiers; checks for conflict.
- **jFrontier():** Checks if a gate with assigned output has x at input; adds to J Frontier list
- **justifyFront():** Finds singular cover for every node in J Frontier, updates and pushes new test cube.
- **D_algorithm_branch():** For a given branch, performs the D Drive and Justification.
- **tcube_intersec():** Given two states, gives corresponding output according to the D-algebra, for intersecting test cubes.
- **D_algorithm():** High level method that first detects faults if any and then calls the D_algorithm_branch function to backtrack and solve conflicts.

  These functions are what we have defined and used in our implementation of the combinational D-Algorithm ATPG. All functions have been documented and commented in the source code. The code for utility functions has been divided into three files: circuit_functions.cpp, d_algo_functions.cpp and headerfiles.cpp, all of which are included as header files in main.cpp. Each of these files includes necessary components and functions for running the code.

4.3 Steps to run the code

The following are the steps to run the submitted code.

- Unzip zip file and go into the folder.
- Run g++ main.cpp to compile the code. Comment out file_read in main.cpp to give your own inputs, or put your input netlist in input.txt (in the specified format) to run the code for the netlist.
- Execute a.out to obtain test vectors for all possible single stuck at faults.

5 Results

5.1 Preliminary circuit testing

We tested our implementation with a simple three gate circuit, which is shown in figure 2.

The wires have been labelled starting from 50, and the nodes have been labelled starting from 0. The input netlist is of the form:

```
8            //Number of nodes
7            //Number of edges
0 4 50       //Format = node1 node2 wirenum
1 4 51
2 5 52
3 5 53
4 6 54
5 6 55
6 7 56
-1           //type of node
-1
-1
-1
```

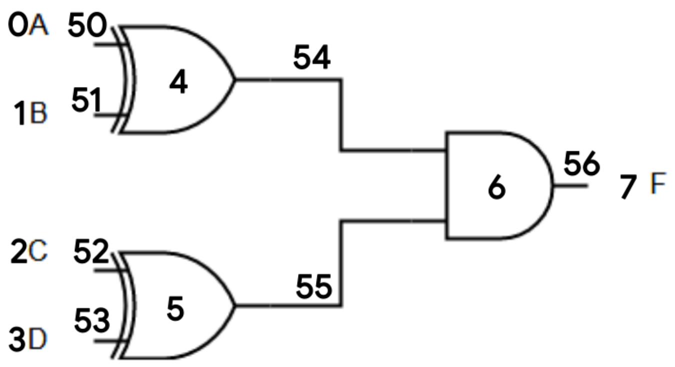

Figure 2: Circuit under test - 1, with labelling

```
5
5
1
-2
```

On giving this in input.txt (without the comments) and running the code, we get the following output.

```
Enter the number of nodes (Branching points, gates, inputs, outputs):
Enter the number of edges (Wires):
Enter connections in netlist format (node1 node2 wirenum):
Enter the type of node:
 -2: output
 -1: input
 0: branching point
 1: AND
 2: OR
 3:NAND
 4:NOR
 5:XOR
 6:XNOR
 7: NOT
0 -> 4 50 ;
1 -> 4 51 ;
2 -> 5 52 ;
3 -> 5 53 ;
4 -> 6 54 ;
5 -> 6 55 ;
6 -> 7 56 ;
7 ->
Node 0 is of type -1
Node 1 is of type -1
Node 2 is of type -1
Node 3 is of type -1
Node 4 is of type 5
Node 5 is of type 5
Node 6 is of type 1
Node 7 is of type -2
Considering s-a-0 for wire 50
Test Complete
Test Vector: 1 1 0 1
Path (Format: Edge-[Node]): 50-[4]-54-[6]-56-[7]-
```

```
Considering s-a-0 for wire 51
Test Complete
Test Vector: 1 1 0 1
Path (Format: Edge-[Node]): 51-[4]-54-[6]-56-[7]-
Considering s-a-0 for wire 52
Test Complete
Test Vector: 0 1 1 1
Path (Format: Edge-[Node]): 52-[5]-55-[6]-56-[7]-
Considering s-a-0 for wire 53
Test Complete
Test Vector: 0 1 1 1
Path (Format: Edge-[Node]): 53-[5]-55-[6]-56-[7]-
Considering s-a-0 for wire 54
Test Complete
Test Vector: 1 0 0 1
Path (Format: Edge-[Node]): 54-[6]-56-[7]-
Considering s-a-0 for wire 55
Test Complete
Test Vector: 0 1 1 0
Path (Format: Edge-[Node]): 55-[6]-56-[7]-
Considering s-a-0 for wire 56
Test Complete
Test Vector: x x x x
Path (Format: Edge-[Node]): 56-[7]-
Considering s-a-1 for wire 50
Test Complete
Test Vector: 0 1 0 1
Path (Format: Edge-[Node]): 50-[4]-54-[6]-56-[7]-
Considering s-a-1 for wire 51
Test Complete
Test Vector: 1 0 0 1
Path (Format: Edge-[Node]): 51-[4]-54-[6]-56-[7]-
Considering s-a-1 for wire 52
Test Complete
Test Vector: 0 1 0 1
Path (Format: Edge-[Node]): 52-[5]-55-[6]-56-[7]-
Considering s-a-1 for wire 53
Test Complete
Test Vector: 0 1 1 0
Path (Format: Edge-[Node]): 53-[5]-55-[6]-56-[7]-
Considering s-a-1 for wire 54
Test Complete
Test Vector: 1 1 0 1
Path (Format: Edge-[Node]): 54-[6]-56-[7]-
Considering s-a-1 for wire 55
Test Complete
Test Vector: 0 1 1 1
Path (Format: Edge-[Node]): 55-[6]-56-[7]-
Considering s-a-1 for wire 56
Test Complete
Test Vector: x x x x
Path (Format: Edge-[Node]): 56-[7]-
```

The graph is constructed using an adjacency list, which is shown to the user after the user is done giving the input. After that, the code simulates every stuck at fault possible on the wires, and calculates a test vector using the d-algorithm and prints it out, along with the path.

5.2 Circuit 2, more complexity

The circuit used here is considerably more complex, with 22 nodes and 28 edges. It is shown in figure 3.

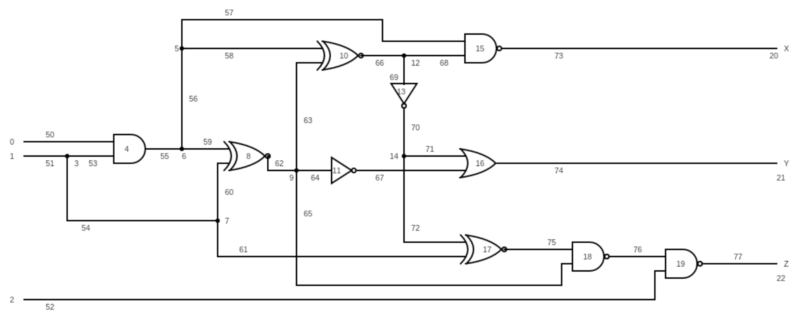

Figure 3: Circuit under test - 2, with labelling

The input is of the form shown below.

```
23
28
0 4 50
1 3 51
2 19 52
3 4 53
3 7 54
4 6 55
6 5 56
5 15 57
5 10 58
6 8 59
7 8 60
7 17 61
8 9 62
9 10 63
9 11 64
9 18 65
10 12 66
11 16 67
12 15 68
12 13 69
13 14 70
14 16 71
14 17 72
15 20 73
16 21 74
17 18 75
18 19 76
19 22 77
-1
-1
-1
0
1
0
0
0
6
0
6
```

```
7
0
7
0
3
2
6
3
3
-2
-2
-2
```

On running this input, our code functions for a brief period of time before crashing due to a memory error caused due to the vectors becoming too big to handle. This will require advanced coding methodologies to tackle, and have not been included in our project. The algorithm however, remains the same.

```
Enter the number of nodes (Branching points, gates, inputs, outputs):
23
Enter the number of edges (Wires):
28
Enter connections in netlist format (node1 node2 wirenum):
0 4 50
1 3 51
2 19 52
3 4 53
3 7 54
4 6 55
6 5 56
5 15 57
5 10 58
6 8 59
7 8 60
7 17 61
8 9 62
9 10 63
9 11 64
9 18 65
10 12 66
11 16 67
12 15 68
12 13 69
13 14 70
14 16 71
14 17 72
15 20 73
16 21 74
17 18 75
18 19 76
19 22 77
Enter the type of node:
 -2: output
 -1: input
 0: branching point
 1: AND
 2: OR
 3:NAND
 4:NOR
 5:XOR
 6:XNOR
 7: NOT
```

```
-1
-1
-1
0
1
0
0
0
6
0
6
7
0
7
0
3
2
6
3
3
-2
-2
-2
0 -> 4 50 ;
1 -> 3 51 ;
2 -> 19 52 ;
3 -> 4 53 ;7 54 ;
4 -> 6 55 ;
5 -> 15 57 ;10 58 ;
6 -> 5 56 ;8 59 ;
7 -> 8 60 ;17 61 ;
8 -> 9 62 ;
9 -> 10 63 ;11 64 ;18 65 ;
10 -> 12 66 ;
11 -> 16 67 ;
12 -> 15 68 ;13 69 ;
13 -> 14 70 ;
14 -> 16 71 ;17 72 ;
15 -> 20 73 ;
16 -> 21 74 ;
17 -> 18 75 ;
18 -> 19 76 ;
19 -> 22 77 ;
20 ->
21 ->
22 ->
Node 0 is of type -1
Node 1 is of type -1
Node 2 is of type -1
Node 3 is of type 0
Node 4 is of type 1
Node 5 is of type 0
Node 6 is of type 0
Node 7 is of type 0
Node 8 is of type 6
Node 9 is of type 0
Node 10 is of type 6
Node 11 is of type 7
Node 12 is of type 0
Node 13 is of type 7
Node 14 is of type 0
Node 15 is of type 3
Node 16 is of type 2
Node 17 is of type 6
Node 18 is of type 3
Node 19 is of type 3
Node 20 is of type -2
Node 21 is of type -2
Node 22 is of type -2
Considering s-a-0 for wire 50
Fault Untestable

Considering s-a-0 for wire 51
Fault Untestable

Considering s-a-0 for wire 52
Test Complete
Test Vector: 1 1 1
Path (Format: Edge-[Node]): 52-[19]-77-[22]-
Considering s-a-0 for wire 53
Fault Untestable

Considering s-a-0 for wire 54
Test Complete
Test Vector: 1 1 1
Path (Format: Edge-[Node]): 54-[7]-61-[17]-75-[18]-76-[19]-77-[22]-
Considering s-a-0 for wire 55
Fault Untestable

Considering s-a-0 for wire 56
Fault Untestable

Considering s-a-0 for wire 57
Fault Untestable

Considering s-a-0 for wire 58
Fault Untestable

Considering s-a-0 for wire 59
Test Complete
Test Vector: 1 1 1
Path (Format: Edge-[Node]): 59-[8]-62-[9]-65-[18]-76-[19]-77-[22]-
Considering s-a-0 for wire 60
Test Complete
Test Vector: 1 1 1
Path (Format: Edge-[Node]): 60-[8]-62-[9]-65-[18]-76-[19]-77-[22]-
Considering s-a-0 for wire 61
Test Complete
Test Vector: 1 1 1
Path (Format: Edge-[Node]): 61-[17]-75-[18]-76-[19]-77-[22]-
Considering s-a-0 for wire 62
Test Complete
Test Vector: 1 1 1
Path (Format: Edge-[Node]): 62-[9]-65-[18]-76-[19]-77-[22]-
Considering s-a-0 for wire 63
Fault Untestable

Considering s-a-0 for wire 64
Fault Untestable

Considering s-a-0 for wire 65
Test Complete
Test Vector: 1 1 1
Path (Format: Edge-[Node]): 65-[18]-76-[19]-77-[22]-
Considering s-a-0 for wire 66
Fault Untestable

Considering s-a-0 for wire 67
free(): invalid next size (fast)
zsh: IOT instruction (core dumped)  ./a.out
```

Our algorithm can however, still work for isolated faults, if the required parts are commented out and the wires are given directly in the code.

5.3 Circuit 3, Complex, but testable

The circuit showcased here is of a slightly reduced complexity, with 16 nodes and 16 edges as shown in figure 4.

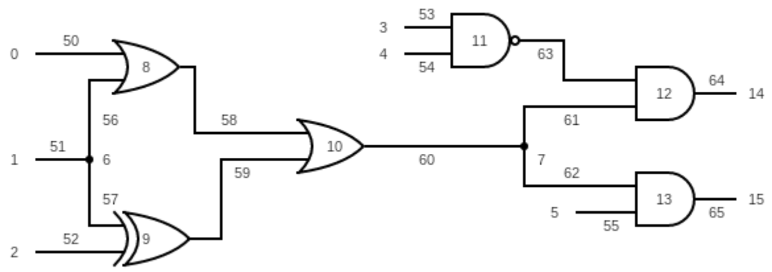

Figure 4: Circuit under test - 3, with labelling

The d-algorithm gives us all possible test vectors for this circuit without crashing, for every possible stuck-at fault. The input is shown below.

```
16
16
0 8 50
1 6 51
2 9 52
3 11 53
4 11 54
5 13 55
6 8 56
6 9 57
7 12 61
7 13 62
8 10 58
9 10 59
10 7 60
11 12 63
12 14 64
13 15 65
-1
-1
-1
-1
-1
-1
0
0
2
```

```
5
2
3
1
1
-2
-2
```

The output for this input is given below.

```
Enter the number of nodes (Branching points, gates, inputs, outputs):
Enter the number of edges (Wires):
Enter connections in netlist format (node1 node2 wirenum):
Enter the type of node:
 -2: output
 -1: input
 0: branching point
 1: AND
 2: OR
 3:NAND
 4:NOR
 5:XOR
 6:XNOR
 7: NOT
0 -> 8 50 ;
1 -> 6 51 ;
2 -> 9 52 ;
3 -> 11 53 ;
4 -> 11 54 ;
5 -> 13 55 ;
6 -> 8 56 ;9 57 ;
7 -> 12 61 ;13 62 ;
8 -> 10 58 ;
9 -> 10 59 ;
10 -> 7 60 ;
11 -> 12 63 ;
12 -> 14 64 ;
13 -> 15 65 ;
14 ->
15 ->
Node 0 is of type -1
Node 1 is of type -1
Node 2 is of type -1
Node 3 is of type -1
Node 4 is of type -1
Node 5 is of type -1
Node 6 is of type 0
Node 7 is of type 0
Node 8 is of type 2
Node 9 is of type 5
Node 10 is of type 2
Node 11 is of type 3
Node 12 is of type 1
Node 13 is of type 1
Node 14 is of type -2
Node 15 is of type -2
Considering s-a-0 for wire 50
Test Complete
Test Vector: 1 0 0 x x 1
Path (Format: Edge-[Node]): 50-[8]-58-[10]-60-[7]-62-[13]-65-[15]-
Considering s-a-0 for wire 51
Fault Untestable
```

```

Considering s-a-0 for wire 52
Test Complete
Test Vector: 0 1 1 x x 1
Path (Format: Edge-[Node]): 52-[9]-59-[10]-60-[7]-62-[13]-65-[15]-
Considering s-a-0 for wire 53
Test Complete
Test Vector: x 1 0 1 1 x
Path (Format: Edge-[Node]): 53-[11]-63-[12]-64-[14]-
Considering s-a-0 for wire 54
Test Complete
Test Vector: x 1 0 1 1 x
Path (Format: Edge-[Node]): 54-[11]-63-[12]-64-[14]-
Considering s-a-0 for wire 55
Test Complete
Test Vector: x 1 0 x x 1
Path (Format: Edge-[Node]): 55-[13]-65-[15]-
Considering s-a-0 for wire 56
Test Complete
Test Vector: 0 1 0 x x 1
Path (Format: Edge-[Node]): 56-[8]-58-[10]-60-[7]-62-[13]-65-[15]-
Considering s-a-0 for wire 57
Test Complete
Test Vector: 0 1 1 x x 1
Path (Format: Edge-[Node]): 57-[9]-59-[10]-60-[7]-62-[13]-65-[15]-
Considering s-a-0 for wire 58
Test Complete
Test Vector: 1 1 0 x x 1
Path (Format: Edge-[Node]): 58-[10]-60-[7]-62-[13]-65-[15]-
Considering s-a-0 for wire 59
Test Complete
Test Vector: 0 0 1 x x 1
Path (Format: Edge-[Node]): 59-[10]-60-[7]-62-[13]-65-[15]-
Considering s-a-0 for wire 60
Test Complete
Test Vector: x 1 x x x 1
Path (Format: Edge-[Node]): 60-[7]-62-[13]-65-[15]-
Considering s-a-0 for wire 61
Test Complete
Test Vector: x 1 0 0 x x
Path (Format: Edge-[Node]): 61-[12]-64-[14]-
Considering s-a-0 for wire 62
Test Complete
Test Vector: x 1 0 x x 1
Path (Format: Edge-[Node]): 62-[13]-65-[15]-
Considering s-a-0 for wire 63
Test Complete
Test Vector: x 1 0 x 0 x
Path (Format: Edge-[Node]): 63-[12]-64-[14]-
Considering s-a-0 for wire 64
Test Complete
Test Vector: x x x x x x
Path (Format: Edge-[Node]): 64-[14]-
Considering s-a-0 for wire 65
Test Complete
Test Vector: x x x x x 1
Path (Format: Edge-[Node]): 65-[15]-
Considering s-a-1 for wire 50
Test Complete
Test Vector: 0 0 0 x x 1
Path (Format: Edge-[Node]): 50-[8]-58-[10]-60-[7]-62-[13]-65-[15]-
```

```
Considering s—a—1 for wire 51
Fault Untestable

Considering s—a—1 for wire 52
Test Complete
Test Vector: 0 1 0 x x 1
Path (Format: Edge−[Node]): 52−[9]−59−[10]−60−[7]−62−[13]−65−[15]−
Considering s—a—1 for wire 53
Test Complete
Test Vector: x 1 0 0 1 x
Path (Format: Edge−[Node]): 53−[11]−63−[12]−64−[14]−
Considering s—a—1 for wire 54
Test Complete
Test Vector: x 1 0 1 0 x
Path (Format: Edge−[Node]): 54−[11]−63−[12]−64−[14]−
Considering s—a—1 for wire 55
Test Complete
Test Vector: x 1 0 x x 0
Path (Format: Edge−[Node]): 55−[13]−65−[15]−
Considering s—a—1 for wire 56
Test Complete
Test Vector: 0 0 0 x x 1
Path (Format: Edge−[Node]): 56−[8]−58−[10]−60−[7]−62−[13]−65−[15]−
Considering s—a—1 for wire 57
Test Complete
Test Vector: 0 0 1 x x 1
Path (Format: Edge−[Node]): 57−[9]−59−[10]−60−[7]−62−[13]−65−[15]−
Considering s—a—1 for wire 58
Test Complete
Test Vector: 0 0 0 x x 1
Path (Format: Edge−[Node]): 58−[10]−60−[7]−62−[13]−65−[15]−
Considering s—a—1 for wire 59
Test Complete
Test Vector: 0 1 1 x x 1
Path (Format: Edge−[Node]): 59−[10]−60−[7]−62−[13]−65−[15]−
Considering s—a—1 for wire 60
Test Complete
Test Vector: 0 0 0 x x 1
Path (Format: Edge−[Node]): 60−[7]−62−[13]−65−[15]−
Considering s—a—1 for wire 61
Test Complete
Test Vector: 0 0 0 0 x x
Path (Format: Edge−[Node]): 61−[12]−64−[14]−
Considering s—a—1 for wire 62
Test Complete
Test Vector: 0 0 0 x x 1
Path (Format: Edge−[Node]): 62−[13]−65−[15]−
Considering s—a—1 for wire 63
Test Complete
Test Vector: x 1 0 1 1 x
Path (Format: Edge−[Node]): 63−[12]−64−[14]−
Considering s—a—1 for wire 64
Test Complete
Test Vector: x x x x x x
Path (Format: Edge−[Node]): 64−[14]−
Considering s—a—1 for wire 65
Test Complete
Test Vector: x x x x x x
Path (Format: Edge−[Node]): 65−[15]−
```

We have successfully simulated all faults in the circuit.

6 Conclusions

The D-algorithm for combinational circuits has been successfully implemented in C++. The circuit is taken in a netlist format, which is parsed and used to construct a graph representation of the circuit, which is then used to simulate faults and to find test vectors to detect the faults using specific algorithms like the D-algorithm in this case. All the code has been documented and understood as well, and hence, the objectives in the problem statement have been accomplished. The input.txt file in the zip has the preliminary circuit used for testing, along with the required code files and output.txt.

7 Further Improvements

The current implementation only works for two input gates. Possible extensions of the work include:

- Extension of D-Algorithm for sequential circuits.
- Addition of functionality for working on gates having more than 2 inputs.

8 Individual Contributions

- L Lakshmanan: Code and Report
- Ananya Sane: Code and Report
- Jaishnav Yarramaneni: Report and Theory
- Eswara Rohan: Report
