aigverse

A Python library for working with logic networks, synthesis, and optimization

Version 0.1.2.dev7

Marcel Walter
Technical University of Munich

Jun 13, 2026

Abstract

`aigverse` is an open-source C++17 and Python infrastructure library for working with logic networks, synthesis, and optimization in Python-first workflows.

It builds directly upon the EPFL Logic Synthesis Libraries, particularly mockturtle, kitty, and lorina, and exposes these mature C++ capabilities through an idiomatic Python interface. This allows users to run synthesis workflows from Python without reimplementing core algorithms there. The result is reusable support for And-Inverter Graph (AIG) construction and manipulation, optimization and equivalence checking, dataset generation, and export into graph and numeric representations for downstream data science and ML pipelines.

Key features include:

- Efficient logic representation using And-Inverter Graphs (AIGs)
- Support for various file formats (AIGER, Verilog, Bench, PLA)
- High-performance C++ backend with a Pythonic interface
- Optional adapters for graph and numeric interoperability in machine learning and data science workflows
- Comprehensive tools for logic synthesis and optimization

`aigverse` is designed as reusable bridge infrastructure rather than as a full end-to-end EDA toolchain.

This documentation provides a comprehensive guide to the `aigverse` library, including {doc} `installation instructions <installation>`, a *quickstart guide*, and detailed {doc} `API documentation <api/aigverse/index>`. The source code of `aigverse` is publicly available on GitHub at marcelwa/aigverse, while pre-built binaries are available via PyPI for all major operating systems and all modern Python versions.

See also

For a deeper dive into the vision and technical details behind `aigverse`, see the presentation **"aigverse: Toward machine learning-driven logic synthesis"** from the Free Silicon Conference (FSiC) 2025. The slides are available on the FSiC wiki and cover the motivation, architecture, and future directions of the `aigverse` project.

Note

A live version of this document is available at aigverse.readthedocs.io.

Contents

I Installation

`aigverse` wraps mature C/C++ synthesis backends from the EPFL Logic Synthesis Libraries with an idiomatic Python interface for Python-first workflows. Optional adapters extend this core with graph and array interoperability for downstream ML and data science pipelines. The resulting Python package is available on PyPI and can be installed on all major operating systems and all active Python versions.

> **Tip**
>
> We highly recommend using uv for working with Python projects. It is an extremely fast Python package and project manager, written in Rust and developed by Astral (the same team behind ruff). It can act as a drop-in replacement for `pip` and `virtualenv`, and provides a more modern and faster alternative to the traditional Python package management tools. It automatically handles the creation of virtual environments and the installation of packages, and is much faster than `pip`. Additionally, it can even set up Python for you if it is not installed yet.
>
> If you do not have uv installed yet, you can install it via:
>
> **macOS and Linux**
>
> ```
> $ curl -LsSf https://astral.sh/uv/install.sh | sh
> ```
>
> **Windows**
>
> ```
> $ powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
> ```
>
> Check out their excellent documentation for more information.

I-A Core Library

To install the core `aigverse` library, you can use `uv` or `pip`.

uv *(recommended)*

```
$ uv pip install aigverse
```

pip

```
(.venv) $ python -m pip install aigverse
```

In most practical cases (under 64-bit Linux, macOS incl. Apple Silicon, and Windows), this requires no compilation and merely downloads and installs a platform-specific pre-built wheel.

Once installed, you can check if the installation was successful by running:

```
(.venv) $ python -c "import aigverse; print(aigverse.__version__)"
```

which should print the installed version of the library.

If you want to use the `aigverse` Python package in your own project, you can simply add it as a dependency to your `pyproject.toml` or `setup.py` file. This will automatically install the `aigverse` package and its dependencies when your project is installed.

uv *(recommended)*

```
$ uv add aigverse
```

pyproject.toml

```toml
[project]
# ...
dependencies = ["aigverse"]
# ...
```

setup.py

```python
from setuptools import setup

setup(
    # ...
    install_requires=["aigverse"],
    # ...
)
```

I-B Machine Learning Adapters

The base installation intentionally excludes ML and data science adapters so that core synthesis workflows remain lightweight and free of heavy optional dependencies. Install the `aigverse[adapters]` extra when you need graph or numeric interoperability in Python ML/data science pipelines:

uv *(recommended)*

```
$ uv pip install "aigverse[adapters]"
```

pip

```
(.venv) $ python -m pip install "aigverse[adapters]"
```

The same syntax applies to adding the `aigverse` package with adapters as a dependency to your own project.

II And-Inverter Graphs (AIGs)

The central data structure for working with And-Inverter Graphs (AIGs) in aigverse is the *Aig* class. It enables efficient representation and manipulation of logic circuits in a form that's well-suited for optimization and verification tasks.

The following will demonstrate how to work with the *Aig* class in Python.

> **Note**
>
> AIGs (And-Inverter Graphs) are a compact representation of Boolean functions using only AND gates and inverters (NOT gates). They are widely used in formal verification, hardware design, and optimization tasks.

II-A Quickstart

The following code snippet demonstrates how to create a simple AIG representing basic logic operations.

```python
from aigverse.networks import Aig

# Create a new AIG network
aig = Aig()

# Create primary inputs
x1 = aig.create_pi()
x2 = aig.create_pi()

# Create logic gates
f_and = aig.create_and(x1, x2)  # AND gate
f_or = aig.create_or(x1, x2)    # OR gate

# Create primary outputs
aig.create_po(f_and)
aig.create_po(f_or)

# Print the size of the AIG network
print(f"AIG Size: {aig.size}")
```

```
AIG Size: 5
```

> **Note**
>
> All primary inputs (PIs) must be created before any logic gates.

II-B Basic AIG Concepts

An AIG consists of nodes and signals:

- **Nodes** represent either primary inputs, constants, or logic gates
- **Signals** reference nodes, possibly with complementation (inversion)

```python
# Create a new AIG
aig = Aig()

# Create a constant (false)
const0 = aig.get_constant(False)
print(f"Constant 0: {const0}")

# Create primary inputs
a = aig.create_pi()
b = aig.create_pi()
print(f"Input a: {a}")
print(f"Input b: {b}")

# Create an AND gate and its complement (NAND)
and_gate = aig.create_and(a, b)
nand_gate = aig.create_not(and_gate)
print(f"AND gate: {and_gate}")
print(f"NAND gate: {nand_gate}")

# Get the node from a signal
node = aig.get_node(and_gate)
print(f"Node of AND gate: {node}")

# Check if a signal is complemented
is_complemented = aig.is_complemented(nand_gate)
print(f"Is NAND complemented? {is_complemented}")
```

```
Constant 0: Signal(0)
Input a: Signal(1)
Input b: Signal(2)
AND gate: Signal(3)
NAND gate: Signal(!3)
Node of AND gate: 3
Is NAND complemented? True
```

Exploring AIG Structure

You can iterate over all nodes in the AIG, or specific subsets like primary inputs or logic gates.

```python
# Create a sample AIG
aig = Aig()
a = aig.create_pi()
b = aig.create_pi()
c = aig.create_pi()
f1 = aig.create_and(a, b)
f2 = aig.create_or(f1, c)
aig.create_po(f2)

# Iterate over all nodes in the AIG
print("All nodes:")
for node in aig.nodes():
    print(f"  Node: {node}")

# Iterate only over primary inputs
print("\nPrimary inputs:")
for pi in aig.pis():
    print(f"  PI: {pi}")

# Iterate only over logic gates
print("\nLogic gates:")
for gate in aig.gates():
    print(f"  Gate: {gate}")

# Iterate over the fanins of a node
print("\nFanins of the OR gate:")
or_node = aig.get_node(f2)
for fanin in aig.fanins(or_node):
    print(f"  Fanin: {fanin}")
```

```
All nodes:
  Node: 0
  Node: 1
  Node: 2
  Node: 3
  Node: 4
  Node: 5

Primary inputs:
  PI: 1
  PI: 2
  PI: 3

Logic gates:
  Gate: 4
  Gate: 5

Fanins of the OR gate:
  Fanin: Signal(!3)
  Fanin: Signal(!4)
```

Building Complex Functions

AIGs support a variety of logic functions beyond just AND gates. Internally, those are decomposed into multiple nodes.

```python
aig = Aig()
a = aig.create_pi()
b = aig.create_pi()
c = aig.create_pi()

# Create various logic functions
and_gate = aig.create_and(a, b)
or_gate = aig.create_or(a, b)
xor_gate = aig.create_xor(a, b)
maj_gate = aig.create_maj(a, b, c)  # Majority function (a&b | a&c | b&c)
ite_gate = aig.create_ite(a, b, c)  # If-then-else: a ? b : c

# Add outputs
aig.create_po(and_gate)
aig.create_po(or_gate)
aig.create_po(xor_gate)
aig.create_po(maj_gate)
aig.create_po(ite_gate)

# Print statistics
print(f"AIG Size: {aig.size}")
print(f"Number of gates: {aig.num_gates}")
print(f"Number of primary inputs: {aig.num_pis}")
print(f"Number of primary outputs: {aig.num_pos}")
```

```
AIG Size: 13
Number of gates: 9
Number of primary inputs: 3
Number of primary outputs: 5
```

II-C AIG Views

AIG views provide alternative representations of AIGs for specific tasks, such as depth computation or fanout analysis. These views can be layered on top of the original AIG, allowing you to work with the same underlying structure while adding additional functionality.

Network and Signal Names

The *NamedAig* class extends the standard AIG with the ability to assign names to the network itself, primary inputs, primary outputs, and internal signals. This is particularly useful for debugging, visualization, and interfacing with external tools that rely on human-readable signal names.

```python
from aigverse.networks import NamedAig

# Create a new named AIG (or construct from existing AIG)
named_aig = NamedAig()

# Set the network name
named_aig.set_network_name("full_adder")

# Create named primary inputs
a = named_aig.create_pi("a")
b = named_aig.create_pi("b")
cin = named_aig.create_pi("cin")

# Create full adder logic
sum = named_aig.create_xor3(a, b, cin)
carry = named_aig.create_maj(a, b, cin)

# Assign names to internal signals
named_aig.set_name(sum, "sum")
named_aig.set_name(carry, "carry")

# Create named primary outputs
named_aig.create_po(sum, "sum_output")
named_aig.create_po(carry, "carry_output")

# Retrieve names
print(f"Network name: {named_aig.get_network_name()}")
print(f"PI names: {[named_aig.get_name(pi) for pi in named_aig.pis()]}")
print(f"Signal name (sum): {named_aig.get_name(sum)}")
print(f"Signal name (carry): {named_aig.get_name(carry)}")
print(f"PO name at index 0: {named_aig.get_output_name(0)}")
print(f"PO name at index 1: {named_aig.get_output_name(1)}")

# Check if a signal has a name
print(f"Has name: {named_aig.has_name(sum)}")
```

```
Network name: full_adder
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 28
     24 named_aig.create_po(carry, "carry_output")
     25
     26 # Retrieve names
     27 print(f"Network name: {named_aig.get_network_name()}")
---> 28 print(f"PI names: {[named_aig.get_name(pi) for pi in named_aig.pis()]}")
     29 print(f"Signal name (sum): {named_aig.get_name(sum)}")
     30 print(f"Signal name (carry): {named_aig.get_name(carry)}")
     31 print(f"PO name at index 0: {named_aig.get_output_name(0)}")

TypeError: get_name(): incompatible function arguments. The following argument types are␣
↪supported:
    1. get_name(self, s: aigverse.networks.AigSignal) -> str

Invoked with types: aigverse.networks.NamedAig, int
```

Named AIGs are automatically created when reading from file formats that contain naming information, e.g., Verilog and AIGER. See *File I/O* for more details.

> **Note**
>
> Names are tied to the specific AIG structure. When you apply optimization algorithms or structural modifications (such as *cleanup_dangling()*), the names will be lost as the return type will be downcast to *Aig*. If preserving names is important, consider reapplying them after optimization.

Depth and Level Computation

The depth of an AIG network represents the longest path from any input to any output, which corresponds to the critical path delay in a circuit. You can compute the depth and level of each node using the *DepthAig* class.

```python
from aigverse.networks import DepthAig

# Create a sample AIG
aig = Aig()
a = aig.create_pi()
b = aig.create_pi()
c = aig.create_pi()
f1 = aig.create_and(a, b)
f2 = aig.create_and(f1, c)
aig.create_po(f2)

# Create a depth view of the AIG
depth_aig = DepthAig(aig)

# Get the depth of the AIG
print(f"Depth of AIG: {depth_aig.num_levels}")

# Print the level of each node
print("\nLevel of each node:")
for node in aig.nodes():
    print(f"  Node {node}: Level {depth_aig.level(node)}")

# Check which nodes are on the critical path
print("\nNodes on critical path:")
for node in aig.nodes():
    if depth_aig.is_on_critical_path(node):
        print(f"  Node {node}")
```

AIGs with Fanout Information

Fanouts of AIG nodes can be collected using *FanoutAig*.

```python
from aigverse.networks import FanoutAig

# Create a sample AIG
aig = Aig()
x1 = aig.create_pi()
x2 = aig.create_pi()
x3 = aig.create_pi()

# Create AND gates
n4 = aig.create_and(x1, x2)
n5 = aig.create_and(n4, x3)
n6 = aig.create_and(n4, n5)
# Create primary outputs
aig.create_po(n6)

fanout_aig = FanoutAig(aig)
print("\nFanout nodes of n4:")
for node in fanout_aig.fanouts(aig.get_node(n4)):
    print(f"  Node {node}")
```

Sequential AIGs

*SequentialAig*s extend standard AIGs to include registers, which allow modeling sequential circuits with memory elements.

```python
from aigverse.networks import SequentialAig

# Create a sequential AIG
seq_aig = SequentialAig()

# Create a primary input and a register output
x = seq_aig.create_pi()       # Regular PI
r = seq_aig.create_ro()       # Register output (sequential PI)

# Create logic
f_and = seq_aig.create_and(x, r)   # AND gate

# Create a primary output and a register input
seq_aig.create_po(f_and)      # Regular PO
seq_aig.create_ri(f_and)      # Register input (sequential PO)

# Print information about the sequential AIG
print(f"Number of PIs: {seq_aig.num_pis}")
print(f"Number of POs: {seq_aig.num_pos}")
print(f"Number of registers: {seq_aig.num_registers}")

# Get the register associations (RI-RO pairs)
print("\nRegister associations:")
registers = seq_aig.registers()
for reg in registers:
    ri, ro = reg
    print(f"  RI: {ri} -> RO: {ro}")
```

Note

When creating sequential AIGs, follow these rules:

1. All register outputs (ROs) must be created after all primary inputs (PIs).
2. All register inputs (RIs) must be created after all primary outputs (POs).
3. As with regular AIGs, all PIs and ROs must be created before any logic gates.

II-D File I/O

AIGs can be read from and written to various file formats.

```python
from aigverse.io import write_aiger, write_verilog, write_dot, read_aiger_into_aig, read_verilog_into_aig, read_pla_into_aig

# Create a sample AIG
aig = Aig()
a = aig.create_pi()
b = aig.create_pi()
f = aig.create_and(a, b)
aig.create_po(f)

# Write to AIGER format
write_aiger(aig, "example.aig")

# Write to Verilog format
write_verilog(aig, "example.v")

# Write to DOT format
write_dot(aig, "example.dot")

# Read from AIGER format
read_aig = read_aiger_into_aig("example.aig")

# Read from Verilog format
read_verilog_aig = read_verilog_into_aig("example.v")

# Read from PLA format
read_pla_aig = read_pla_into_aig("example.pla")

print(f"Original AIG size: {aig.size}")
print(f"Read AIGER AIG size: {read_aig.size}")
print(f"Read Verilog AIG size: {read_verilog_aig.size}")
print(f"Read PLA AIG size: {read_pla_aig.size}")
```

> **Note**
>
> The gate-level Verilog file support constitutes a very small subset of the Verilog standard, similar in extent to what ABC supports. For more information, see the `lorina` parser used by this project.

II-E Index Lists

Alternatively, index lists provide a compact, serialization-friendly representation of an AIG's structure as a flat list of integers. This is useful for ML pipelines, dataset generation, or exporting AIGs for use in environments where fixed-size numeric arrays are required.

```python
from aigverse.networks import AigIndexList

# Create a sample AIG
aig = Aig()
a = aig.create_pi()
b = aig.create_pi()
c = aig.create_pi()
d = aig.create_pi()
t0 = aig.create_and(a, b)
t1 = aig.create_and(~c, ~d)
t2 = aig.create_xor(t0, t1)
aig.create_po(t2)

# Convert an AIG to an index list
indices = aig.to_index_list()

# Convert an index list back to an AIG
aig2 = indices.to_aig()

# Convert to a Python list
indices = [int(i) for i in indices]
print(indices)
```

The first three entries encode number of PIs, number of POs, and number of gates, respectively. In the example above, those are 4, 1, and 5. Successive pairs of indices refer to the fanins signals of nodes. Each fanin exists in two polarities: negated = odd index, and non-negated = even index. In the example, 2 and 4 refer to the non-negated signals originating from PIs 1 and 2. These form the first AND gate. The subsequent 7 and 9 are odd, hence, negated. If the first index is lower than the second, an AND gate is encoded. Otherwise, it is an XOR gate. The final indices of the list refer to the PO signals. It must be ensured that they match the encoded number of POs.

For more information on the index list format, see `mockturtle`'s documentation.

II-F `pickle` Support

AIGs support Python's `pickle` protocol, allowing you to serialize and deserialize AIG objects for persistent storage. This is useful for saving intermediate results, sharing AIGs between processes, quickly restoring previously computed networks, or integrating with Python-first data science and machine learning workflows.

```python
import pickle

# Save AIG to a file using pickle
with open("aig.pkl", "wb") as f:
    pickle.dump(aig, f)

# Load the AIG from the pickle file
with open("aig.pkl", "rb") as f:
    unpickled_aig = pickle.load(f)
```

You can also pickle and unpickle multiple AIGs at once by storing them in a tuple or list.

```python
aig1 = Aig()
a1 = aig1.create_pi()
b1 = aig1.create_pi()
f1 = aig1.create_and(a1, b1)
aig1.create_po(f1)

aig2 = Aig()
a2 = aig2.create_pi()
b2 = aig2.create_pi()
f2 = aig2.create_or(a2, b2)
aig2.create_po(f2)

# Pickle both AIGs together
with open("aigs.pkl", "wb") as f:
    pickle.dump((aig1, aig2), f)

# Unpickle them
with open("aigs.pkl", "rb") as f:
    unpickled_aig1, unpickled_aig2 = pickle.load(f)
```

III Truth Tables

The aigverse library provides support for working with truth tables, which are a fundamental representation of Boolean functions. The `TruthTable` class offers efficient manipulation and analysis of Boolean functions.

> **Note**
>
> Truth tables provide a complete specification of a Boolean function by listing all possible input combinations and their corresponding outputs. They are particularly useful for small functions where exhaustive enumeration is feasible.

III-A Creating Truth Tables

Truth tables can be created in several ways:

```python
from aigverse.utils import TruthTable

# Initialize a truth table with 3 variables (2^3 = 8 entries)
tt = TruthTable(3)

# Create a truth table from a hex string (representing the Majority function)
tt.create_from_hex_string("e8")
print(f"Truth table from hex string: {tt.to_binary()}")

# Create a truth table for an AND function
tt_and = TruthTable(2)
tt_and.create_from_hex_string("8")
print(f"AND function: {tt_and.to_binary()}")

# Create a truth table for an OR function
tt_or = TruthTable(2)
tt_or.create_from_hex_string("e")
print(f"OR function: {tt_or.to_binary()}")
```

```
Truth table from hex string: 11101000
AND function: 1000
OR function: 1110
```

III-B Basic Manipulation

Truth tables provide various methods for bit manipulation:

```python
# Create a truth table
tt = TruthTable(3)
tt.create_from_hex_string("e8")  # Majority function

# Get individual bits
print(f"Original truth table: {tt.to_binary()}")
print(f"Bit at position 0: {int(tt.get_bit(0))}")
print(f"Bit at position 7: {int(tt.get_bit(7))}")

# Flip bits
tt.flip_bit(0)
tt.flip_bit(7)
print(f"After flipping bits 0 and 7: {tt.to_binary()}")

# Clear the truth table
tt.clear()
print(f"After clearing: {tt.to_binary()}")

# Check if constant
print(f"Is constant 0? {tt.is_const0()}")
```

```
Original truth table: 11101000
Bit at position 0: 0
Bit at position 7: 1
After flipping bits 0 and 7: 01101001
After clearing: 00000000
Is constant 0? True
```

III-C Truth Table Properties

You can analyze various properties of truth tables:

```python
# Create a truth table for XOR
tt_xor = TruthTable(2)
tt_xor.create_from_hex_string("6")  # XOR function
print(f"XOR function: {tt_xor.to_binary()}")

# Get number of variables and bits
print(f"Number of variables: {tt_xor.num_vars()}")
print(f"Number of bits: {tt_xor.num_bits()}")

# Check if the function is balanced (equal number of 0s and 1s)
num_ones = sum(int(tt_xor.get_bit(i)) for i in range(tt_xor.num_bits()))
is_balanced = num_ones == tt_xor.num_bits() // 2
print(f"Is balanced? {is_balanced}")
```

```
XOR function: 0110
Number of variables: 2
Number of bits: 4
Is balanced? True
```

III-D Truth Table Simulation

The simulation of AIGs and other logic networks using truth tables is covered in the *Simulation section* of the Algorithms documentation. This approach allows you to obtain the truth tables for outputs and internal nodes of a logic network.

III-E `pickle` Support

Truth tables support Python's `pickle` protocol, allowing you to serialize and deserialize them for persistent storage or use in data science workflows.

```python
import pickle

# Create a truth table
tt = TruthTable(3)
tt.create_from_hex_string("d8")  # ITE function

# Pickle the truth table
with open("tt.pkl", "wb") as f:
    pickle.dump(tt, f)

# Unpickle the truth table
with open("tt.pkl", "rb") as f:
    unpickled_tt = pickle.load(f)

# Verify that the unpickled object is identical
print(f"Original:    {tt.to_binary()}")
print(f"Unpickled:   {unpickled_tt.to_binary()}")
print(f"Equivalent:  {tt == unpickled_tt}")
```

```
Original:    11011000
Unpickled:   11011000
Equivalent:  True
```

You can also pickle multiple truth tables at once by storing them in a list or tuple.

IV Algorithms

This section covers the various algorithms available in aigverse for working with And-Inverter Graphs (AIGs) and other logic representations. These algorithms enable simulation, optimization, and verification of logic networks.

IV-A Simulation

Simulation algorithms allow you to evaluate the outputs of a logic network for all possible input combinations, effectively generating truth tables for the network's outputs and internal nodes.

Functional Simulation

For simulating AIGs with truth tables, the *simulate()* and {py:func} `~aigverse.algorithms.simulate_nodes` functions allow you to obtain truth tables for outputs and internal nodes of an AIG.

```python
from aigverse.networks import Aig
from aigverse.algorithms import simulate, simulate_nodes

# Create a sample AIG
aig = Aig()
a = aig.create_pi()
b = aig.create_pi()
f_and = aig.create_and(a, b)
f_or = aig.create_or(a, b)
aig.create_po(f_and)
aig.create_po(f_or)

# Simulate the outputs
output_tts = simulate(aig)

# Print the truth tables
print("Truth tables of outputs:")
for i, tt in enumerate(output_tts):
    print(f"  Output {i}: {tt.to_binary()}")

# Simulate all nodes
node_tts = simulate_nodes(aig)

# Print the truth table of each node
print("\nTruth tables of nodes:")
for node, tt in node_tts.items():
    print(f"  Node {node}: {tt.to_binary()}")
```

```
Truth tables of outputs:
  Output 0: 1000
  Output 1: 1110

Truth tables of nodes:
  Node 4: 0001
  Node 3: 1000
  Node 2: 1100
  Node 1: 1010
  Node 0: 0000
```

IV-B Optimization

AIG optimization aims to reduce the number of AND gates and inverters in a circuit while maintaining its logical functionality. Different optimization techniques target various aspects of the AIG structure.

Basic Optimization Workflow

The typical optimization workflow involves:

1. Creating or loading an AIG
2. Applying one or more optimization algorithms
3. Verifying correctness through equivalence checking

```python
from aigverse.io import read_aiger_into_aig

# Load the i10 benchmark circuit - a real-world example
aig = read_aiger_into_aig("examples/i10.aig")

# Print statistics about the loaded circuit
print(f"i10 benchmark:")
print(f"  I/O: {aig.num_pis}/{aig.num_pos}")
print(f"  AND gates: {aig.num_gates}")
```

```
i10 benchmark:
  I/O: 257/224
  AND gates: 2675
```

Resubstitution

Resubstitution identifies portions of logic that can be expressed using existing signals in the network. This technique is particularly effective at identifying and eliminating redundant logic.

```python
from aigverse.algorithms import aig_resubstitution

# Clone the AIG for comparison
aig_resub = aig.clone()

# Apply resubstitution
aig_resub = aig_resubstitution(aig_resub, window_size=12)

print(f"Original AND gates: {aig.num_gates}")
print(f"After resubstitution: {aig_resub.num_gates} AND gates")
print(f"Reduction: {aig.num_gates - aig_resub.num_gates} gates ({(aig.num_gates - aig_resub.
↪num_gates) / aig.num_gates * 100:.2f}%)")
```

```
Original AND gates: 2675
After resubstitution: 2173 AND gates
Reduction: 502 gates (18.77%)
```

Sum-of-Products Refactoring

SOP (Sum of Products) refactoring collapses parts of the AIG into truth tables, then re-synthesizes those portions using Sum-of-Products representations. This can find more efficient implementations for complex logic functions.

```python
from aigverse.algorithms import sop_refactoring

# Clone the AIG for comparison
aig_refactor = aig.clone()

# Apply SOP refactoring
aig_refactor = sop_refactoring(aig_refactor, use_reconvergence_cut=True)

print(f"Original AND gates: {aig.num_gates}")
print(f"After SOP refactoring: {aig_refactor.num_gates} AND gates")
print(f"Reduction: {aig.num_gates - aig_refactor.num_gates} gates ({(aig.num_gates - aig_refactor.num_gates) / aig.num_gates * 100:.2f}%)")
```

```
Original AND gates: 2675
After SOP refactoring: 2239 AND gates
Reduction: 436 gates (16.30%)
```

Cut Rewriting

Cut rewriting identifies small subgraphs (cuts) in the AIG and replaces them with pre-computed optimal implementations from a library. This technique leverages NPN-equivalence classes to find the best possible implementation for each cut.

```python
from aigverse.algorithms import aig_cut_rewriting

# Clone the AIG for comparison
aig_rewrite = aig.clone()

# Apply cut rewriting
aig_rewrite = aig_cut_rewriting(aig_rewrite, cut_size=4)

print(f"Original AND gates: {aig.num_gates}")
print(f"After cut rewriting: {aig_rewrite.num_gates} AND gates")
print(f"Reduction: {aig.num_gates - aig_rewrite.num_gates} gates ({(aig.num_gates - aig_rewrite.num_gates) / aig.num_gates * 100:.2f}%)")
```

```
Original AND gates: 2675
After cut rewriting: 2200 AND gates
Reduction: 475 gates (17.76%)
```

Balancing

Balancing performs (E)SOP factoring to minimize the number of levels in the AIG.

```python
from aigverse.algorithms import balancing
from aigverse.networks import DepthAig

# Clone the AIG for comparison
aig_balance = aig.clone()

# Apply balancing
aig_balance = balancing(aig_balance, rebalance_function="sop")

# Compute depth
original_depth = DepthAig(aig).num_levels
balanced_depth = DepthAig(aig_balance).num_levels

print(f"Original depth: {original_depth} levels")
print(f"After balancing: {balanced_depth} levels")
print(f"Reduction in depth: {original_depth - balanced_depth} levels ({(original_depth - balanced_depth) / original_depth * 100:.2f}%)")
```

```
Original depth: 50 levels
After balancing: 32 levels
Reduction in depth: 18 levels (36.00%)
```

Combining Optimization Techniques

For best results, optimization techniques are typically applied in combination, often in multiple passes. The order of application can significantly impact the final result.

```
# Apply optimization techniques in sequence
aig_opt = aig.clone()

# First pass
aig_opt = aig_resubstitution(aig_opt)
aig_opt = sop_refactoring(aig_opt)
aig_opt = aig_cut_rewriting(aig_opt)

# Second pass
aig_opt = aig_resubstitution(aig_opt)
aig_opt = sop_refactoring(aig_opt)

print(f"\nTotal optimization results:")
print(f"- Original: {aig.num_gates} AND gates")
print(f"- Optimized: {aig_opt.num_gates} AND gates")
print(f"- Total reduction: {aig.num_gates - aig_opt.num_gates} gates ({(aig.num_gates - aig_
↪opt.num_gates) / aig.num_gates * 100:.2f}%)")
```

```
Total optimization results:
- Original: 2675 AND gates
- Optimized: 1950 AND gates
- Total reduction: 725 gates (27.10%)
```

Some algorithms offer the `inplace=True` keyword argument for performance-sensitive pipelines of chained optimization. Calling functions such as *`aig_resubstitution()`* and *`sop_refactoring()`* with `inplace=True` mutates the passed network and returns `None`:

```
from aigverse.algorithms import cleanup_dangling

aig_fast = aig.clone()
aig_resubstitution(aig_fast, inplace=True)
sop_refactoring(aig_fast, inplace=True)

# Explicit cleanup step after in-place chaining
aig_fast = cleanup_dangling(aig_fast)
```

> **Note**
>
> When choosing this route, users are responsible to call *`cleanup_dangling()`* to obtain a structurally valid AIG.

IV-C Equivalence Checking

Equivalence checking algorithms verify that two logic networks implement the same function, which is especially important after performing optimizations.

```
from aigverse.algorithms import equivalence_checking

# Verify that our optimized circuit from the previous section maintains functional equivalence
are_equivalent = equivalence_checking(aig, aig_opt)
print(f"\nOriginal and optimized benchmark circuits are equivalent: {are_equivalent}")
print(f"This confirms our optimization preserved the circuit's functionality while reducing")
print(f"the gate count from {aig.num_gates} to {aig_opt.num_gates} AND gates.")
```

```
Original and optimized benchmark circuits are equivalent: True
This confirms our optimization preserved the circuit's functionality while reducing
the gate count from 2086 to 1950 AND gates.
```

V Generators

The module provides generation helpers for reproducible random dataset creation and structured arithmetic/control benchmarks.

V-A Random AIG Generation

Use *random_aig()* for reproducible one-shot generation.

```python
from aigverse.generators import random_aig

# One-shot random AIG with fixed size
single = random_aig(num_pis=4, num_gates=12, seed=42)
print(single.num_pis, single.num_gates)
```

```
4 12
```

V-B Python-Side Batching

Batching is intentionally kept on the Python side to keep experiment orchestration reproducible and easy to compose with downstream data science and ML code. For fixed-size datasets, call *random_aig()* repeatedly with fixed `num_pis` and `num_gates`.

> **Note**
>
> *random_aig()* does not guarantee a fixed `num_pos` across seeds: mockturtle's random topology can leave a different number of dangling nodes depending on the sampled structure. As a result, repeated calls with identical `num_pis`/`num_gates` may still produce different output counts if the seed varies. For ML training loops, when you need a uniform label shape, you can filter generated examples by `num_pos` or resample seeds until `num_pos` matches your target.

```python
from aigverse.generators import random_aig

batch_fixed = [random_aig(num_pis=4, num_gates=12, seed=1000 + i) for i in range(8)]
print(len(batch_fixed), batch_fixed[0].num_pis, batch_fixed[0].num_gates)
```

```
8 4 12
```

For size-diverse datasets, sample sizes in Python and pass them to *random_aig()*.

```python
import random

from aigverse.generators import random_aig

rng = random.Random(7)

dataset = [
  random_aig(
    num_pis=rng.randint(3, 5),
    num_gates=rng.randint(10, 16),
    seed=2000 + i,
  )
  for i in range(8)
]

print([(aig.num_pis, aig.num_gates) for aig in dataset])
```

```
[(4, 11), (4, 15), (3, 10), (5, 10), (4, 14), (3, 14), (3, 10), (3, 13)]
```

V-C Structured Benchmark Builders

Complete benchmark networks of arithmetic and control circuits of arbitrary bitwidth can be created via respective high-level generator functions.

```python
from aigverse.generators import ripple_carry_adder, carry_lookahead_adder

rca4 = ripple_carry_adder(bitwidth=4)
print(f"RCA: I/O: {rca4.num_pis}/{rca4.num_pos}, gates: {rca4.num_gates}")
cla4 = carry_lookahead_adder(bitwidth=4)
print(f"CLA: I/O: {cla4.num_pis}/{cla4.num_pos}, gates: {cla4.num_gates}")
```

```
RCA: I/O: 8/5, gates: 24
CLA: I/O: 8/5, gates: 36
```

V-D Control Generators

Control builders follow the same high-level model.

```python
from aigverse.generators import binary_decoder, multiplexer

mux = multiplexer(bitwidth=4)
print(f"MUX: I/O: {mux.num_pis}/{mux.num_pos}, gates: {mux.num_gates}")

decoder = binary_decoder(num_select_bits=4)
print(f"Decoder: I/O: {decoder.num_pis}/{decoder.num_pos}, gates: {decoder.num_gates}")
```

```
MUX: I/O: 9/4, gates: 12
Decoder: I/O: 4/16, gates: 24
```

Available structured builders include *ripple_carry_adder()*, *carry_lookahead_adder()*, *sideways_sum_adder()*, *ripple_carry_multiplier()*, *multiplexer()*, and *binary_decoder()*.

VI Machine Learning Integration

Many ML and data science workflows are Python-first, while mature logic synthesis data structures and algorithms are predominantly implemented in C/C++ toolchains. Without reusable infrastructure, projects often reimplement synthesis functionality in Python or rely on brittle wrappers and file-conversion scripts around external tools. `aigverse` addresses this gap by exposing native AIG construction, manipulation, optimization, and analysis through an idiomatic Python API, and by providing optional adapters for graph and numeric representations. This enables reproducible dataset generation, feature extraction, and downstream model experimentation from within standard Python workflows.

VI-A Adapters

Adapters are the interoperability layer for ML and data science workflows. To keep the base library lightweight, they are optional and installed via the `adapters` extra only when needed. See the *Installation* documentation for more details.

VI-B Dataset Generation

The generators module offers a reproducible way to create synthetic AIG datasets before converting them into ML-ready formats.

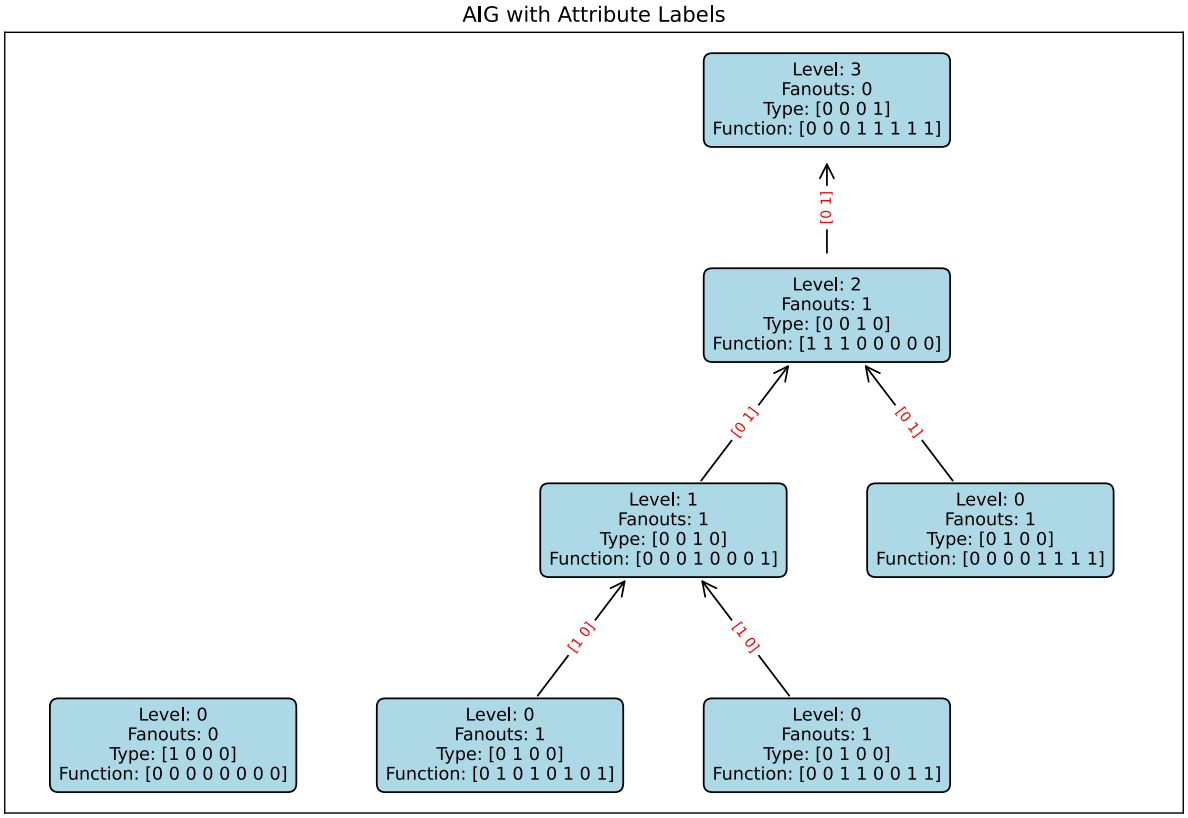
```python
import random

from aigverse.generators import random_aig

rng = random.Random(1234)

# Generate a reproducible, size-diverse AIG dataset
dataset = [
    random_aig(
        num_pis=rng.randint(4, 6),
        num_gates=rng.randint(20, 40),
        seed=5000 + i,
    )
    for i in range(16)
]

print(len(dataset), dataset[0].num_pis, dataset[0].num_gates)
```

```
16 5 23
```

NetworkX

The NetworkX adapter allows you to convert an AIG into a `DiGraph` object. This enables use of graph-analysis and graph-learning tooling that operates on NetworkX graphs. Once converted, you can extract node and edge features, visualize the structure (e.g., with Matplotlib), or feed the graph into downstream model pipelines.

```python
import matplotlib.pyplot as plt
import networkx as nx
from networkx.drawing.nx_agraph import graphviz_layout
import numpy as np

from aigverse.networks import Aig
import aigverse.adapters

# Create a sample AIG
aig = Aig()
a = aig.create_pi()
b = aig.create_pi()
c = aig.create_pi()
f1 = aig.create_and(a, b)
f2 = aig.create_or(f1, c)
aig.create_po(f2)

# Convert the AIG to a NetworkX graph
G = aig.to_networkx(levels=True, fanouts=True, node_tts=True, dtype=np.int32)

# Generate the initial layout using Graphviz's 'dot' program
pos = graphviz_layout(G, prog="dot")

# Invert the y-axis to flip the layout upside down
# This places the primary inputs (level 0) at the bottom
for node, position in pos.items():
    pos[node] = (position[0], -position[1])

# Prepare the labels for nodes and edges from graph attributes
node_labels = {
    node:␣
↪f"Level: {data['level']}\nFanouts: {data['fanouts']}\nType: {data['type']}\nFunction: {data['function']}"
    for node, data in G.nodes(data=True)
}
edge_labels = {(u, v): data["type"] for u, v, data in G.edges(data=True)}

# Plot the graph
plt.figure(figsize=(12, 8))
plt.title("AIG with Attribute Labels")

# Draw the graph structure (just the edges and arrows)
nx.draw_networkx_nodes(G, pos, node_size=0)

# Draw the node labels with a bounding box
nx.draw_networkx_labels(
    G,
    pos,
    labels=node_labels,
    font_size=10,
    bbox={"facecolor": "lightblue", "edgecolor": "black", "boxstyle": "round,pad=0.5"},
)

# Draw the network edges
nx.draw_networkx_edges(
    G,
    pos,
    node_size=5000,
    arrows=True,
    arrowstyle="->",
    arrowsize=20,
)

# Draw the edge labels to show edge attributes
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color="red", font_size=8)

# Pad and show the plot
plt.margins(x=0.2)
plt.show()
```



VI-C Truth Tables

Truth tables can be easily converted to Python lists or NumPy arrays, making them compatible with standard ML libraries such as scikit-learn, PyTorch, or TensorFlow. Since `TruthTable` objects are iterable, this conversion is direct and intuitive. You can use these arrays as labels or features in supervised learning tasks, or as part of a dataset for training and evaluating models.

```python
from aigverse.utils import TruthTable
import numpy as np

# Create a simple truth table, e.g., a 3-input majority function
tt = TruthTable(3)
tt.create_from_hex_string("e8")

# Export to a list
tt_list = list(tt)
print(f"As list: {tt_list}")

# Export to NumPy arrays of different types
tt_np_bool = np.array(tt)
print(f"As NumPy bool array:  {tt_np_bool}")
tt_np_int = np.array(tt, dtype=np.int32)
print(f"As NumPy int array:   {tt_np_int}")
tt_np_float = np.array(tt, dtype=np.float64)
print(f"As NumPy float array: {tt_np_float}")


# These arrays can now be used as labels for an ML model.
# For example, let's generate the corresponding feature matrix:
def generate_inputs(num_vars):
    inputs = []
    for i in range(2**num_vars):
        # Convert i to binary and pad with zeros
        binary = bin(i)[2:].zfill(num_vars)
        inputs.append([int(bit) for bit in binary])
    return np.array(inputs)


feature_matrix = generate_inputs(tt.num_vars())
labels = tt_np_int  # Using the integer array as labels

print("\nFeature matrix (X) and labels (y) for ML:")
print("X:\n", feature_matrix)
print("y:\n", labels)
```

```
As list: [False, False, False, True, False, True, True, True]
As NumPy bool array:  [False False False  True False  True  True  True]
As NumPy int array:   [0 0 0 1 0 1 1 1]
As NumPy float array: [0. 0. 0. 1. 0. 1. 1. 1.]

Feature matrix (X) and labels (y) for ML:
X:
 [[0 0 0]
 [0 0 1]
 [0 1 0]
 [0 1 1]
 [1 0 0]
 [1 0 1]
 [1 1 0]
 [1 1 1]]
y:
 [0 0 0 1 0 1 1 1]
```

VII aigverse

aigverse: A Python Library for Logic Networks, Synthesis, and Optimization.

VII-A Submodules

aigverse.adapters

aigverse adapters for ML tasks.

Submodules

aigverse.adapters.networkx AIG to NetworkX adapter.

Module Contents

to_networkx(*self:* Aig, ** (Keyword-only parameters separator (PEP 3102))*, *levels:* *bool* = *False*, *fanouts:* *bool* = *False*, *node_tts:* *bool* = *False*, *graph_tts:* *bool* = *False*, *dtype:* type*[generic]* = *np.int8*) → DiGraph

Converts an `Aig` to a `DiGraph`.

This function transforms the AIG into a directed graph representation using the NetworkX library. It allows for the inclusion of various attributes for the graph, its nodes, and edges, making it suitable for graph-based machine learning tasks.

Note that the constant-0 node is always included in the graph, as index 0, even if it is not referenced by any edges.

Parameters

- **self** – The AIG object to convert.
- **levels** – If True, computes and adds level information for each node and the total number of levels to the graph, as attributes `level` and `levels`, respectively. Defaults to False.
- **fanouts** – If True, adds the fanout count for each node as an integer `fanouts` attribute (`0` for synthetic PO nodes). Defaults to False.
- **node_tts** – If True, computes and adds a truth table for each node as a `function` attribute. Defaults to False.
- **graph_tts** – If True, computes and adds the graph's overall truth table as a `function` attribute to the graph. Defaults to False.
- **dtype** – The data type for truth tables and all one-hot encodings. Defaults to `int8`. For machine learning tasks, a floating-point type such as `float32` or `float64` may be more appropriate, as it allows for gradient-based optimization.

Returns A `DiGraph` representing the AIG.

Graph Attributes:

- type (str): `"AIG"`.
- num_pis (int): Number of primary inputs.
- num_pos (int): Number of primary outputs.
- num_gates (int): Number of AND gates.
- levels (int, optional): Total number of levels in the AIG.
- function (list[`ndarray`], optional): Graph's truth tables.
- name (str, optional): Network name (only for `NamedAig`).

Node Attributes:

- index (int): The node's identifier.
- level (int, optional): The level of the node in the AIG.
- fanouts (int, optional): Fanout count of the node.
- function (`ndarray`, optional): The node's truth table.
- type (`ndarray`): One-hot encoded vector representing the node type (`const`, `pi`, `gate`, `po`). The data type is determined by the `dtype` argument, defaulting to `int8`.

Edge Attributes:

- type (`ndarray`): One-hot encoded vector representing the edge type (`regular`, `inverted`). The data type is determined by the `dtype` argument, defaulting to `int8`.
- name (str, optional): Signal name for `NamedAig` primary output name for edges to synthetic PO nodes.

aigverse.algorithms

Provides synthesis and optimization algorithms for logic network types.

Module Contents

equivalence_checking(*spec:* Aig, *impl:* Aig, ***, *conflict_limit:* *int* = *0*, *functional_reduction:* *bool* = *True*, *verbose:* *bool* = *False*) → bool | None

Checks functional equivalence between a specification and implementation network using SAT solving.

Parameters

- **spec** – The specification network.

- **impl** – The implementation network.
- **conflict_limit** – SAT conflict limit. A value of `0` means no limit.
- **functional_reduction** – Whether to perform functional reduction of the miter before checking.
- **verbose** – Whether to print verbose progress output.

Returns `True` if equivalent, `False` if not equivalent, or `None` if the procedure did not finish before the configured limit.

Raises **RuntimeError** – If miter construction fails due to incompatible interfaces (PI/PO count mismatch).

cleanup_dangling(*ntk:* Aig, **, remove_dangling_pis: bool = False, remove_redundant_pos: bool = False*) → *Aig*

Removes dangling logic (dead nodes) from a network.

Parameters

- **ntk** – The input logic network.
- **remove_dangling_pis** – Whether to also remove dangling primary inputs.
- **remove_redundant_pos** – Whether to remove redundant primary outputs.

Returns A cleaned network with dangling structures removed.

sop_refactoring(*ntk:* Aig, **, max_pis: int = 6, allow_zero_gain: bool = False, use_reconvergence_cut: bool = False, use_dont_cares: bool = False, use_quick_factoring: bool = True, try_both_polarities: bool = True, consider_inverter_cost: bool = False, verbose: bool = False, inplace: bool = False*) → *Aig* | None

Performs SOP-based network refactoring.

Parameters

- **ntk** – The input logic network.
- **max_pis** – Maximum number of leaves used in local windows.
- **allow_zero_gain** – Whether substitutions with zero gain are allowed.
- **use_reconvergence_cut** – Whether to use reconvergence-driven cuts.
- **use_dont_cares** – Whether to use don't-care information.
- **use_quick_factoring** – Whether to use the quick SOP factoring heuristic.
- **try_both_polarities** – Whether both output polarities are explored.
- **consider_inverter_cost** – Whether inverter cost is included in optimization.
- **verbose** – Whether to print verbose progress output.
- **inplace** – Whether to mutate `ntk` in place.

Returns The refactored network if `inplace` is `False`. Otherwise `None`.

Raises **RuntimeError** – If refactoring fails in the underlying synthesis engine.

aig_resubstitution(*ntk:* Aig, **, max_pis: int = 8, max_divisors: int = 150, max_inserts: int = 2, skip_fanout_limit_for_roots: int = 1000, skip_fanout_limit_for_divisors: int = 100, verbose: bool = False, use_dont_cares: bool = False, window_size: int = 12, preserve_depth: bool = False, inplace: bool = False*) → *Aig* | None

Performs AIG resubstitution-based optimization.

Parameters

- **ntk** – The input logic network.
- **max_pis** – Maximum number of leaves in a local window.
- **max_divisors** – Maximum number of candidate divisors.
- **max_inserts** – Maximum number of inserted nodes per replacement.
- **skip_fanout_limit_for_roots** – Fanout threshold to skip root candidates.
- **skip_fanout_limit_for_divisors** – Fanout threshold to skip divisor candidates.
- **verbose** – Whether to print verbose progress output.
- **use_dont_cares** – Whether to use don't-care information.
- **window_size** – Window size used for don't-care computation.
- **preserve_depth** – Whether replacements must preserve depth.
- **inplace** – Whether to mutate `ntk` in place.

Returns The optimized network if `inplace` is `False`. Otherwise `None`.

aig_cut_rewriting(*ntk:* Aig, **, cut_size: int = 4, cut_limit: int = 8, minimize_truth_table: bool = True, allow_zero_gain: bool = False, use_dont_cares: bool = False, min_cand_cut_size: int = 3, min_cand_cut_size_override: int | None = None, preserve_depth: bool = False, verbose: bool = False, very_verbose: bool = False*) → *Aig*

Rewrites an AIG network using cut-based NPN resynthesis.

Parameters

- **ntk** – The input logic network.
- **cut_size** – Maximum cut size used during cut enumeration.
- **cut_limit** – Maximum number of cuts retained per node.
- **minimize_truth_table** – Whether to minimize cut truth tables.
- **allow_zero_gain** – Whether replacements with zero gain are allowed.
- **use_dont_cares** – Whether to use don't-care information.
- **min_cand_cut_size** – Minimum candidate cut size.
- **min_cand_cut_size_override** – Optional override for minimum candidate cut size.
- **preserve_depth** – Whether replacements must preserve network depth.
- **verbose** – Whether to print verbose progress output.
- **very_verbose** – Whether to print highly detailed progress output.

Returns A rewritten network.

balancing(*ntk:* Aig, **, cut_size: int = 4, cut_limit: int = 8, minimize_truth_table: bool = True, only_on_critical_path: bool = False, rebalance_function: Literal['sop', 'esop'] = 'sop', sop_both_phases: bool = True, verbose: bool = False*) → *Aig*

Balances a network using SOP or ESOP-based local restructuring.

Parameters

- **ntk** – The input logic network.
- **cut_size** – Maximum cut size used during cut enumeration.
- **cut_limit** – Maximum number of cuts retained per node.
- **minimize_truth_table** – Whether to minimize cut truth tables during enumeration.
- **only_on_critical_path** – Whether to balance only nodes on the critical path.
- **rebalance_function** – Rebalancing engine to use. Supported values are `"sop"` and `"esop"`.
- **sop_both_phases** – Whether to consider both phases in SOP/ESOP balancing.
- **verbose** – Whether to print verbose progress output.

Returns A new balanced network.

Raises **ValueError** – If `rebalance_function` is not one of the supported values.

simulate(*ntk:* Aig) → list[*TruthTable*]

Simulates all primary outputs of a network as truth tables.

Parameters: **ntk** – The input logic network.

Returns A list containing one truth table per primary output.

Raises **MemoryError** – If the truth tables cannot be allocated due to memory limits.

simulate_nodes(*ntk:* Aig) → dict[int, *TruthTable*]

Simulates all nodes of a network as truth tables.

Parameters: **ntk** – The input logic network.

Returns A dictionary that maps node identifiers to truth tables.

Raises **MemoryError** – If the truth tables cannot be allocated due to memory limits.

aigverse.generators

Provides generators for random and structured benchmark construction.

Module Contents

random_aig(**, num_pis: int, num_gates: int, seed: int = 3405688830*) → *Aig*

Generates a single random AIG with a fixed size.

Parameters

- **num_pis** – Number of primary inputs.
- **num_gates** – Number of logic gates.

- **seed** – Seed controlling random choices.

Returns A randomly generated AIG.

Raises

- **ValueError** – If `num_pis` or `num_gates` is `0`.
- **TypeError** – If `num_pis` or `num_gates` cannot be converted to `uint32`.

ripple_carry_adder(*bitwidth: int*) → *Aig*

Creates a complete ripple-carry adder benchmark network.

Parameters **bitwidth** – Number of bits per operand.

Returns An `Aig` with `2 * bitwidth` primary inputs and `bitwidth + 1` primary outputs (sum plus carry-out).

Raises **ValueError** – If `bitwidth` is not greater than `0`.

carry_lookahead_adder(*bitwidth: int*) → *Aig*

Creates a complete carry-lookahead adder benchmark network.

Parameters **bitwidth** – Number of bits per operand.

Returns An `Aig` with `2 * bitwidth` primary inputs and `bitwidth + 1` primary outputs (sum plus carry-out).

Raises **ValueError** – If `bitwidth` is not greater than `0`.

ripple_carry_multiplier(*bitwidth: int*) → *Aig*

Creates a complete ripple-carry multiplier benchmark network.

Parameters **bitwidth** – Number of bits per operand.

Returns An `Aig` with `2 * bitwidth` primary inputs and `2 * bitwidth` primary outputs representing the product bits.

Raises **ValueError** – If `bitwidth` is not greater than `0`.

sideways_sum_adder(*bitwidth: int*) → *Aig*

Creates a complete sideways sum adder benchmark network.

Parameters **bitwidth** – Number of input bits.

Returns An `Aig` with `bitwidth` primary inputs and output bits encoding the population count of the input word.

Raises **ValueError** – If `bitwidth` is not greater than `0`.

multiplexer(*bitwidth: int*) → *Aig*

Creates a complete word-level n-bit 2:1 MUX network.

Parameters **bitwidth** – Number of bits in each data input word.

Returns An `Aig` with `1 + 2 * bitwidth` primary inputs and `bitwidth` primary outputs.

Raises **ValueError** – If `bitwidth` is not greater than `0`.

binary_decoder(*num_select_bits: int*) → *Aig*

Creates a complete binary-decoder network.

Parameters **num_select_bits** – Number of select input bits.

Returns An `Aig` with `num_select_bits` primary inputs and `2 ** num_select_bits` primary outputs.

Raises **ValueError** – If `num_select_bits` is not greater than `0`.

aigverse.io

Provides file import and export functions for logic networks.

The module contains readers and writers for common file formats in the domain.

Module Contents

read_aiger_into_aig(*filename: str | PathLike*) → *NamedAig*

Reads a binary AIGER file into a logic network.

Parameters **filename** – Path to the AIGER file.

Returns The parsed network instance.

Raises **RuntimeError** – If parsing the AIGER file fails.

read_ascii_aiger_into_aig(*filename: str | PathLike*) → *NamedAig*

Reads an ASCII AIGER file into a logic network.

Parameters **filename** – Path to the ASCII AIGER file.
Returns The parsed network instance.
Raises **RuntimeError** – If parsing the ASCII AIGER file fails.

read_aiger_into_sequential_aig(*filename: str | PathLike*) → *SequentialAig*

Reads a binary AIGER file into a logic network.

Parameters **filename** – Path to the AIGER file.
Returns The parsed network instance.
Raises **RuntimeError** – If parsing the AIGER file fails.

read_ascii_aiger_into_sequential_aig(*filename: str | PathLike*) → *SequentialAig*

Reads an ASCII AIGER file into a logic network.

Parameters **filename** – Path to the ASCII AIGER file.
Returns The parsed network instance.
Raises **RuntimeError** – If parsing the ASCII AIGER file fails.

write_aiger(*ntk:* Aig, *filename: str | PathLike*) → None

Writes a logic network to a binary AIGER file.

Parameters

- **ntk** – The network to serialize.
- **filename** – Destination path for the AIGER file.

read_pla_into_aig(*filename: str | PathLike*) → *Aig*

Reads a PLA file into a logic network.

Parameters **filename** – Path to the PLA file.
Returns The parsed network instance.
Raises **RuntimeError** – If parsing the PLA file fails.

read_verilog_into_aig(*filename: str | PathLike*) → *NamedAig*

Reads a synthesized gate-level Verilog netlist into a logic network.

Parameters **filename** – Path to the Verilog file.
Returns The parsed network instance.
Raises **RuntimeError** – If parsing the Verilog file fails.

write_verilog(*ntk:* Aig, *filename: str | PathLike*) → None

Writes a logic network to a Verilog netlist.

Parameters

- **ntk** – The network to serialize.
- **filename** – Destination path for the Verilog file.

write_dot(*ntk:* Aig, *filename: str | PathLike*) → None

Writes a logic network to a Graphviz DOT file for visualization.

Parameters

- **ntk** – The network to serialize.
- **filename** – Destination path for the DOT file.

aigverse.networks

Provides logic network data structures and derived views.

This module includes network types, edge and index list utilities, and helper objects for structural manipulation.

Module Contents

class AigSignal(*index: int*, *complement: bool*)

Represents a signal in an AIG.

Signals point to nodes and may be complemented.

property index: int

Node index referenced by the signal.

property complement: bool

Whether this signal is complemented.

property data: int

Raw packed signal representation.

__hash__() → int

Returns a hash value for dictionary/set usage.

__eq__(*other: object*) → bool

Returns whether two signals are equal.

Parameter **other** – Object to compare.
Returns `True` if equal, otherwise `False`.

__ne__(*other: object*) → bool

Returns whether two signals are not equal.

Parameter **other** – Object to compare.
Returns `True` if not equal, otherwise `False`.

__lt__(*other:* AigSignal) → bool

Returns whether this signal is ordered before another signal.

__invert__() → *AigSignal*

Returns the complemented signal.

__pos__() → *AigSignal*

Returns a normalized positive-phase signal.

__neg__() → *AigSignal*

Returns a normalized negative-phase signal.

__xor__(*complement: bool*) → *AigSignal*

XORs the signal phase with a Boolean complement bit.

Parameter **complement** – Complement bit to XOR with the current signal phase.
Returns A phase-adjusted signal.

class Aig

Represents an AIG and its structural operations.

> **Note**
>
> to_index_list() keeps combinational structure only. Augmented view metadata is not preserved.

clone() → *Aig*

Creates a structural copy of the network.

property size: int

Number of nodes in the network.

property num_gates: int

Number of logic gates in the network.

property num_pis: int

Number of primary inputs.

property num_pos: int

Number of primary outputs.

get_node(*s:* AigSignal) → int

Returns the node referenced by a signal.

make_signal(*n: int*) → *AigSignal*

Creates a signal from a node.

is_complemented(*s:* AigSignal) → bool

Returns whether a signal is complemented.

node_to_index(*n: int*) → int

Returns the integer index of a node.

index_to_node(*index: int*) → int

Returns the node for an index.

pi_index(*n: int*) → int

Returns the primary-input position of a node.

pi_at(*index: int*) → int

Returns the primary input node at `index`.

po_index(*s:* AigSignal) → int

Returns the primary-output position of a signal.

po_at(*index: int*) → *AigSignal*

Returns the primary output signal at `index`.

get_constant(*value: bool*) → *AigSignal*

Returns the constant signal for a Boolean value.

create_pi() → *AigSignal*

Creates and returns a new primary input signal.

create_po(*f:* AigSignal) → int

Creates a primary output from signal `f`.

property is_combinational: bool

Whether the network is combinational.

create_buf(*a:* AigSignal) → *AigSignal*

Creates a buffer.

create_not(*a:* AigSignal) → *AigSignal*

Creates an inversion.

create_and(*a:* AigSignal, *b:* AigSignal) → *AigSignal*

Creates an AND.

create_nand(*a:* AigSignal, *b:* AigSignal) → *AigSignal*

Creates a NAND.

create_or(*a:* AigSignal, *b:* AigSignal) → *AigSignal*

Creates an OR.

create_nor(*a:* AigSignal, *b:* AigSignal) → *AigSignal*

Creates a NOR.

create_xor(*a:* AigSignal, *b:* AigSignal) → *AigSignal*

Creates an XOR.

create_xnor(*a:* AigSignal, *b:* AigSignal) → *AigSignal*

Creates an XNOR.

create_lt(*a:* AigSignal, *b:* AigSignal) → *AigSignal*

Creates a less-than comparator.

create_le(*a:* AigSignal, *b:* AigSignal) → *AigSignal*

Creates a less-or-equal comparator.

create_maj(*a:* AigSignal, *b:* AigSignal, *c:* AigSignal) → *AigSignal*

Creates a majority.

create_ite(*cond:* AigSignal, *f_then:* AigSignal, *f_else:* AigSignal) → *AigSignal*

Creates an if-then-else.

create_xor3(*a:* AigSignal, *b:* AigSignal, *c:* AigSignal) → *AigSignal*

Creates a 3-input XOR.

create_nary_and(*fs: Sequence[*AigSignal*]*) → *AigSignal*

Creates an n-ary AND.

create_nary_or(*fs: Sequence[*AigSignal*]*) → *AigSignal*

Creates an n-ary OR.

create_nary_xor(*fs: Sequence[*AigSignal*]*) → *AigSignal*

Creates an n-ary XOR.

clone_node(*other:* Aig, *source: int*, *children: Sequence[*AigSignal*]*) → *AigSignal*

Clones one node from `other` into this network.

nodes() → list[int]

Returns a list of all nodes in order of creation.

gates() → list[int]

Returns a list of all non-constant and non-PI nodes in order of creation.

pis() → list[int]

Returns a list of all primary input nodes in order of creation.

pos() → list[*AigSignal*]

Returns a list of all primary output signals in order of creation.

fanins(*n: int*) → list[*AigSignal*]

Returns fanin signals of node `n`.

fanin_size(*n: int*) → int

Returns the number of fanins of node `n`.

fanout_size(*n: int*) → int

Returns the number of fanouts of node `n`.

is_constant(*n: int*) → bool

Returns whether `n` is a constant node.

is_pi(*n: int*) → bool

Returns whether `n` is a primary input.

has_and(*a:* AigSignal, *b:* AigSignal) → *AigSignal* | None

Returns whether an AND with fanins `a` and `b` already exists.

is_and(*n: int*) → bool

Returns whether `n` is an AND node.

is_or(*n: int*) → bool

Returns whether `n` is an OR node.

is_xor(*n: int*) → bool

Returns whether `n` is an XOR node.

is_maj(*n: int*) → bool

Returns whether `n` is a majority node.

is_ite(*n: int*) → bool

Returns whether `n` is an if-then-else node.

is_xor3(*n: int*) → bool

Returns whether `n` is a 3-input XOR node.

is_nary_and(*n: int*) → bool

Returns whether `n` is an n-ary AND node.

is_nary_or(*n: int*) → bool

Returns whether `n` is an n-ary OR node.

to_edge_list(*regular_weight: int = 0*, *inverted_weight: int = 1*) → *AigEdgeList*

Converts the network to an edge list.

Parameters

- **regular_weight** – Weight assigned to non-inverted edges.
- **inverted_weight** – Weight assigned to inverted edges.

Returns The corresponding edge-list representation.

to_index_list() → *AigIndexList*

Converts the network to an index-list encoding.

Returns The corresponding index-list representation.

to_networkx(*, *levels: bool = False*, *fanouts: bool = False*, *node_tts: bool = False*, *graph_tts: bool = False*, *dtype:* type*[generic] = ...*) → DiGraph

Converts an `Aig` to a `DiGraph`.

This function transforms the AIG into a directed graph representation using the NetworkX library. It allows for the inclusion of various attributes for the graph, its nodes, and edges, making it suitable for graph-based machine learning tasks.

Note that the constant-0 node is always included in the graph, as index 0, even if it is not referenced by any edges.

Parameters

- **self** – The AIG object to convert.
- **levels** – If True, computes and adds level information for each node and the total number of levels to the graph, as attributes `level` and `levels`, respectively. Defaults to False.
- **fanouts** – If True, adds the fanout count for each node as an integer `fanouts` attribute (`0` for synthetic PO nodes). Defaults to False.
- **node_tts** – If True, computes and adds a truth table for each node as a `function` attribute. Defaults to False.
- **graph_tts** – If True, computes and adds the graph's overall truth table as a `function` attribute to the graph. Defaults to False.

- **dtype** – The data type for truth tables and all one-hot encodings. Defaults to `int8`. For machine learning tasks, a floating-point type such as `float32` or `float64` may be more appropriate, as it allows for gradient-based optimization.

Returns A `DiGraph` representing the AIG.

Graph Attributes:

- type (str): `"AIG"`.
- num_pis (int): Number of primary inputs.
- num_pos (int): Number of primary outputs.
- num_gates (int): Number of AND gates.
- levels (int, optional): Total number of levels in the AIG.
- function (list[`ndarray`], optional): Graph's truth tables.
- name (str, optional): Network name (only for `NamedAig`).

Node Attributes:

- index (int): The node's identifier.
- level (int, optional): The level of the node in the AIG.
- fanouts (list, optional): Fanout node indices. Present if `add_fanouts=True`.
- function (`ndarray`, optional): The node's truth table.
- type (`ndarray`): The node type (`const`, `pi`, `gate`, `po`). The data type is determined by the `dtype` argument, defaulting to `int8`.

Edge Attributes:

- type (`ndarray`): The edge type (`regular`, `inverted`). The data type is determined by the `dtype` argument, defaulting to `int8`.
- name (str, optional): Signal name for edges to primary output nodes.

__len__() → int

Returns the number of nodes.

__iter__() → Iterator[int]

Returns an iterator over all nodes.

__contains__(*value: object*) → bool

Returns whether a node index is contained in the network.

__bool__() → bool

Returns `True` for non-trivial networks.

__copy__() → *Aig*

Returns a shallow structural copy.

__deepcopy__(*memo: dict*) → *Aig*

Returns a deep structural copy.

__setstate__(*state: object*) → None

Restores a network from a pickled state tuple.

Parameters **state** – Tuple containing one index-list payload.
Raises **ValueError** – If the state shape or payload is invalid.

__getstate__() → tuple

Returns pickle state as an index-list tuple.

Preserves only combinational structure and does not capture augmented view metadata.

class NamedAig

class NamedAig(*ntk:* NamedAig)

class NamedAig(*ntk:* Aig)

Bases: *Aig*

Extends a network with input/output and node names.

clone() → *NamedAig*

Creates a structural copy including names.

__copy__() → *NamedAig*

Returns a shallow copy of the named view.

__deepcopy__(*memo: dict*) → *NamedAig*

Returns a deep copy of the named view.

create_pi(*name: str = ''*) → *AigSignal*

Creates a primary input with an optional name.

create_po(*f:* AigSignal, *name: str = ''*) → int

Creates a primary output with an optional name.

set_network_name(*name: str*) → None

Sets the network name.

get_network_name() → str

Returns the network name.

has_name(*s:* AigSignal) → bool

Returns whether signal `s` has a name.

set_name(*s:* AigSignal, *name: str*) → None

Assigns a name to signal `s`.

get_name(*s:* AigSignal) → str

Returns the name of signal `s`.

has_output_name(*index: int*) → bool

Returns whether output `index` has a name.

set_output_name(*index: int*, *name: str*) → None

Sets the name of output `index`.

get_output_name(*index: int*) → str

Returns the name of output `index`.

class DepthAig

class DepthAig(*ntk:* DepthAig)

class DepthAig(*ntk:* Aig)

Bases: *Aig*

Extends a network with depth information.

clone() → *DepthAig*

Creates a copy of the depth view.

__copy__() → *DepthAig*

Returns a shallow copy of the depth view.

__deepcopy__(*memo: dict*) → *DepthAig*

Returns a deep copy of the depth view.

property num_levels: int

Current network depth in levels.

level(*n: int*) → int

Returns the level of node `n`.

is_on_critical_path(*n: int*) → bool

Returns whether node `n` is on at least one critical path.

update_levels() → None

Recomputes level information.

create_po(*f:* AigSignal) → int

Creates an output and updates depth information.

class FanoutAig

class FanoutAig(*ntk:* Aig)

class FanoutAig(*ntk:* FanoutAig)

Bases: *Aig*

Extends a network with fanout information.

clone() → *FanoutAig*

Creates a structural copy of the fanout view.

__copy__() → *FanoutAig*

Returns a shallow copy of the fanout view.

__deepcopy__(*memo: dict*) → *FanoutAig*

Returns a deep copy of the fanout view.

fanouts(*n: int*) → list[int]

Returns fanout nodes of node `n`.

class AigRegister

class AigRegister(*register:* AigRegister)

Represents metadata for one sequential register.

property control: str

Optional control signal.

property init: int

Initial value of the register.

property type: str

Register type/category.

class SequentialAig

Bases: *Aig*

Represents a sequential network with register interfaces.

clone() → *SequentialAig*

Creates a structural copy of the sequential network.

__copy__() → *SequentialAig*

Returns a shallow copy of the sequential network.

__deepcopy__(*memo: dict*) → *SequentialAig*

Returns a deep copy of the sequential network.

create_pi() → *AigSignal*

Creates a primary input.

create_po(*f:* AigSignal) → int

Creates a primary output.

create_ro() → *AigSignal*

Creates a register output node.

create_ri(*f:* AigSignal) → int

Creates a register input signal.

property is_combinational: bool

Whether the network is combinational.

is_ci(*n: int*) → bool

Returns whether `n` is a combinational input.

is_pi(*n: int*) → bool

Returns whether `n` is a primary input.

is_ro(*n: int*) → bool

Returns whether `n` is a register output.

property num_pis: int

Number of primary inputs.

property num_pos: int

Number of primary outputs.

property num_cis: int

Number of combinational inputs.

property num_cos: int

Number of combinational outputs.

property num_registers: int

Number of registers.

pi_at(*index: int*) → int

Returns primary input at `index`.

po_at(*index: int*) → *AigSignal*

Returns primary output at `index`.

ci_at(*index: int*) → int

Returns combinational input at `index`.

co_at(*index: int*) → *AigSignal*

Returns combinational output at `index`.

ro_at(*index: int*) → int

Returns register output at `index`.

ri_at(*index: int*) → *AigSignal*

Returns register input at `index`.

set_register(*index: int*, *reg:* AigRegister) → None

Sets metadata for register `index`.

register_at(*index: int*) → *AigRegister*

Returns metadata for register `index`.

pi_index(*n: int*) → int

Returns PI index of node `n`.

ci_index(*n: int*) → int

Returns CI index of node `n`.

co_index(*s:* AigSignal) → int

Returns CO index of signal `s`.

ro_index(*n: int*) → int

Returns RO index of node `n`.

ri_index(*s:* AigSignal) → int

Returns RI index of signal `s`.

ro_to_ri(*s:* AigSignal) → *AigSignal*

Maps a register output signal to its input.

ri_to_ro(*s:* AigSignal) → int

Maps a register input signal to its output.

to_index_list() → NoReturn

Sequential networks cannot be encoded as combinational index lists.

__getstate__() → NoReturn

Sequential networks are not pickleable via combinational index-list state.

__setstate__(*state: object*) → NoReturn

Sequential networks cannot be restored from combinational index-list state.

to_edge_list(*regular_weight: int = 0*, *inverted_weight: int = 1*) → *AigEdgeList*

Converts the sequential network to an edge list.

pis() → list[int]

Returns all primary input nodes.

pos() → list[*AigSignal*]

Returns all primary output signals.

cis() → list[int]

Returns all combinational input nodes.

cos() → list[*AigSignal*]

Returns all combinational output signals.

ros() → list[int]

Returns all register output nodes.

ris() → list[*AigSignal*]

Returns all register input signals.

registers() → list[tuple[*AigSignal*, int]]

Returns all register pairs as `(ri_signal, ro_node)` tuples.

class AigEdge

class AigEdge(*source: int*, *target: int*, *weight: int = 0*)

Represents a directed edge in a logic network graph. A weight attribute may encode inversion.

property source: int

Source node identifier.

property target: int

Target node identifier.

property weight: int

Edge weight value.

__eq__(*other: object*) → bool

Checks whether two edges are equal.

Parameters **other** – Object to compare.
Returns `True` if `other` is an edge with equal fields, otherwise `False`.

__ne__(*other: object*) → bool

Checks whether two edges are not equal.

Parameters **other** – Object to compare.
Returns `True` if `other` is not equal to this edge.

class AigEdgeList

class AigEdgeList(*ntk:* Aig)

class AigEdgeList(*edges: Sequence[*AigEdge*]*)

class AigEdgeList(*ntk:* Aig, *edges: Sequence[*AigEdge*]*)

Represents a list of edges associated with a network.

property ntk: *Aig*

Underlying network associated with this list.

property edges: list[*AigEdge*]

Stored edges in insertion order.

append(*edge:* AigEdge) → None

Appends an edge to the list.

Parameters **edge** – Edge to append.

clear() → None

Removes all edges from the list.

__iter__() → Iterator[*AigEdge*]

Returns an iterator over stored edges.

__len__() → int

Returns the number of edges.

__getitem__(*index: int*) → *AigEdge*

Returns the edge at a given position.

Parameters **index** – Edge index. Negative indices are supported.
Returns The edge at the requested position.
Raises **IndexError** – If `index` is out of bounds.

__setitem__(*index: int*, *edge:* AigEdge) → None

Replaces the edge at a given position.

Parameters

- **index** – Edge index. Negative indices are supported.
- **edge** – Replacement edge.

Raises **IndexError** – If `index` is out of bounds.

class AigIndexList(*num_pis: int = 0*)

class AigIndexList(*values: Sequence[int]*)

Represents an index-list encoding of an AIG network.

raw() → list[int]

Returns the raw integer encoding.

Returns A list of encoded index-list values.

property size: int

Number of raw entries in the encoding.

property num_gates: int

Number of encoded gates.

property num_pis: int

Number of encoded primary inputs.

property num_pos: int

Number of encoded primary outputs.

add_inputs(*n: int = 1*) → None

Appends primary inputs to the encoding.

Parameters **n** – Number of inputs to append.

add_and(*lit0: int*, *lit1: int*) → int

Appends an AND gate to the encoding.

Parameters

- **lit0** – First fanin literal.
- **lit1** – Second fanin literal.

add_xor(*lit0: int*, *lit1: int*) → int

Appends an XOR gate to the encoding.

Parameters

- **lit0** – First fanin literal.
- **lit1** – Second fanin literal.

add_output(*lit: int*) → None

Appends a primary output literal.

Parameters **lit** – Output literal.

clear() → None

Clears all encoded data.

to_aig() → *Aig*

Decodes the index list into an AIG network.

Returns A decoded AIG network.

gates() → list[tuple[int, int]]

Returns encoded gate fanins as literal pairs.

Returns A list of `(lit0, lit1)` tuples for each gate.

pos() → list[int]

Returns encoded primary output literals.

Returns A list of output literals.

__iter__() → Iterator[int]

Returns an iterator over the raw encoding values.

__getitem__(*index: int*) → int

Returns one raw encoding value by index.

Parameters **index** – Position in the raw encoding.
Returns The raw value at `index`.
Raises **IndexError** – If `index` is out of range.

__setitem__(*index: int*, *value: int*) → None

Sets one raw encoding value by index.

Parameters

- **index** – Position in the raw encoding.
- **value** – Replacement raw value.

Raises **IndexError** – If `index` is out of range.

__len__() → int

Returns the number of raw encoding entries.

aigverse.utils

Provides utility data structures and functions.

Module Contents

class TruthTable(*num_vars: int*)

Represents a dynamic Boolean truth table.

__eq__(*other: object*) → bool

Checks equality with another truth table.

__ne__(*other: object*) → bool

Checks inequality with another truth table.

__lt__(*other:* TruthTable) → bool

Lexicographically compares two truth tables.

__and__(*other:* TruthTable) → *TruthTable*

Computes bitwise AND with another truth table.

__or__(*other:* TruthTable) → *TruthTable*

Computes bitwise OR with another truth table.

__xor__(*other:* TruthTable) → *TruthTable*

Computes bitwise XOR with another truth table.

__invert__() → *TruthTable*

Computes bitwise NOT.

__len__() → int

Returns the number of bits.

__getitem__(*index: int*) → bool

Returns one bit by index.

Parameters**index** – Bit index.
Returns The bit value.
Raises **IndexError** – If `index` is out of range.

__setitem__(*index: int*, *value: bool*) → None

Sets one bit by index.

Parameters

- **index** – Bit index.
- **value** – New bit value.

Raises **IndexError** – If `index` is out of range.

__iter__() → Iterator[bool]

Returns an iterator over all bits.

num_vars() → int

Returns the number of variables.

num_blocks() → int

Returns the number of storage blocks.

num_bits() → int

Returns the number of truth table bits.

__copy__() → *TruthTable*

Returns a shallow copy of the truth table.

__deepcopy__(*arg:* *dict*, */ (Positional-only parameter separator (PEP 570))*) → *TruthTable*

Returns a deep copy of the truth table.

__assign__(*other:* TruthTable) → *TruthTable*

Assigns from another truth table with a compatible shape.

Parameters **other** – Source truth table.
Returns The updated truth table.

__hash__() → int

Returns a hash value for dictionary/set usage.

__getstate__() → tuple

Returns pickle state as `(num_vars, words)`.

__setstate__(*state:* *tuple*) → None

Restores a truth table from pickle state.

Parameters **state** – Tuple `(num_vars, words)`.
Raises

- **RuntimeError** – If the serialized state is malformed.
- **TypeError** – If nanobind cannot convert the pickle payload to the expected C++ types.

set_bit(*index:* *int*) → None

Sets one bit to `1`.

Parameters **index** – Bit index.
Raises **IndexError** – If `index` is out of range.

get_bit(*index:* *int*) → bool

Returns one bit value.

Parameters **index** – Bit index.
Returns The bit value.
Raises **IndexError** – If `index` is out of range.

clear_bit(*index:* *int*) → None

Clears one bit to `0`.

Parameters **index** – Bit index.
Raises **IndexError** – If `index` is out of range.

flip_bit(*index:* *int*) → None

Toggles one bit value.

Parameters **index** – Bit index.
Raises **IndexError** – If `index` is out of range.

get_block(*block_index:* *int*) → int

Returns one 64-bit storage block.

Parameters **block_index** – Block index.
Returns The block value.
Raises **IndexError** – If `block_index` is out of range.

create_nth_var(*var_index: int*, *complement: bool = False*) → None

Creates the projection function for one variable.

Parameters

- **var_index** – Variable index to project.
- **complement** – Whether to complement the projection.

Raises **IndexError** – If `var_index` is out of range.

create_from_binary_string(*binary: str*) → None

Loads bit values from a binary string.

Parameters **binary** – Binary string of length `num_bits`.

Raises **ValueError** – If the string length does not match `num_bits`.

create_from_hex_string(*hexadecimal: str*) → None

Loads bit values from a hexadecimal string.

Parameters **hexadecimal** – Hex string matching the truth table size.

Raises **ValueError** – If the string length does not match the expected size.

create_random() → None

Fills the truth table with random bits.

create_majority() → None

Fills the truth table with the majority function.

clear() → None

Clears all bits to `0`.

count_ones() → int

Returns the number of set bits.

count_zeroes() → int

Returns the number of cleared bits.

is_const0() → bool

Returns whether all bits are `0`.

is_const1() → bool

Returns whether all bits are `1`.

to_binary() → str

Returns the truth table as a binary string.

to_hex() → str

Returns the truth table as a hexadecimal string.

ternary_majority(*a:* TruthTable, *b:* TruthTable, *c:* TruthTable) → *TruthTable*

Computes the ternary majority of three truth tables.

Parameters

- **a** – First truth table.
- **b** – Second truth table.
- **c** – Third truth table.

Returns The bitwise majority truth table.

cofactor0(*tt:* TruthTable, *var_index: int*) → *TruthTable*

Computes the cofactor with respect to assigning one variable to `0`.

Parameters

- **tt** – Input truth table.
- **var_index** – Index of the variable to cofactor.

Returns The cofactored truth table with `var_index` fixed to `0`.

Raises **ValueError** – If `var_index` is out of range.

cofactor1(*tt:* TruthTable, *var_index: int*) → *TruthTable*

Computes the cofactor with respect to assigning one variable to 1.

Parameters

- **tt** – Input truth table.
- **var_index** – Index of the variable to cofactor.

Returns The cofactored truth table with `var_index` fixed to 1.

Raises **ValueError** – If `var_index` is out of range.

VII-B Package Contents

__version__: str

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

W