bfcl

Release 1.0.1

Nth Party, Ltd.

Jul 08, 2022

CONTENTS

Bristol Fashion Circuit Library (BFCL) for working with circuit definitions represented using the Bristol Fashion.

CHAPTER

ONE

PURPOSE

This library includes data structures and associated methods for working with logical circuits typically used in secure multi-party computation (MPC) applications. The data structures follow in their organization the Bristol Fashion format, extrapolating and generalizing where necessary in order to support a wider variety of features and operations.

CHAPTER TWO

PACKAGE INSTALLATION AND USAGE

The package is available on PyPI:

```
python -m pip install bfcl
```

The library can be imported in the usual way:

```
import bfcl
from bfcl import *
```

This library makes it possible to parse a circuit definition that conforms to the Bristol Fashion syntax:

```
>>> ss = ['7 36', '2 4 4', '1 1']
>>> ss += ['2 1 0 1 15 AND', '2 1 2 3 16 AND']
>>> ss += ['2 1 15 16 8 AND', '2 1 4 5 22 AND']
>>> ss += ['2 1 6 7 23 AND', '2 1 22 23 9 AND']
>>> ss += ['2 1 8 9 35 AND']
>>> c = bfc('\n'.join(ss))
```

A string representation that conforms to the Bristol Fashion syntax can be emitted:

```
>>> for line in c.emit().split('\n'):
    ...     print(line)
    ...
    7 36
    2 4 4
    1 1
    2 1 0 1 15 AND
    2 1 2 3 16 AND
    2 1 15 16 8 AND
    2 1 4 5 22 AND
    2 1 6 7 23 AND
    2 1 22 23 9 AND
    2 1 8 9 35 AND
```

It is possible to evaluate a circuit on a sequence of input bit vectors. The circuit defined in the example above takes two 4-bit input vectors and returns the logical conjunction of all the bits. In the example below, it is evaluated on a few pairs of input bit vectors. The result is organized into a list of output bit vectors according to the original circuit definition (in the example below, the result consists of only a single output bit vector that contains a single bit):

```
>>> c.evaluate([[1, 0, 1, 1], [1, 1, 1, 0]])
[[0]]
```

(continues on next page)

(continued from previous page)

```
>>> c.evaluate([[1, 1, 1, 1], [1, 1, 1, 1]])
[[1]]
```

As an alternative to using a string representation to define a circuit, it is also possible to construct a circuit using the circuit library. In the example below, the constructor for the `circuit` class found in the bfcl library is applied to an object built using the classes and methods exported by the circuit library (note the use of a synonym to avoid a conflict with the `circuit` class defined in the bfcl library):

```
>>> import circuit as circuit_
>>> c = circuit_.circuit()
>>> g0 = c.gate(circuit_.op.id_, is_input=True)
>>> g1 = c.gate(circuit_.op.id_, is_input=True)
>>> g2 = c.gate(circuit_.op.and_, [g0, g1])
>>> g3 = c.gate(circuit_.op.id_, [g2], is_output=True)
>>> bfc(c).emit().split('\n')
['2 4', '1 2', '1 1', '2 1 0 1 2 AND', '1 1 2 3 LID']
```

CHAPTER THREE

DOCUMENTATION

The library contains one module:

3.1 bfcl module

Python library for working with circuit definitions represented using the Bristol Fashion.

class bfcl.bfcl.**operation**(*iterable=()*, */*)

Bases: `logical.logical.logical`

Data structure for an individual gate operation. This class is derived from the `logical` class exported by the logical library. This module indirectly imports the `logical` class via the `op` synonym defined in the circuit library. See the documentation for the `logical` class for more information on this data structure and how logical operations are represented as tuples of integers.

token_op_pairs = [('LID', (0, 1)), ('INV', (1, 0)), ('FLS', (0, 0, 0, 0)), ('AND', (0, 0, 0, 1)), ('NIM', (0, 0, 1, 0)), ('FST', (0, 0, 1, 1)), ('NIF', (0, 1, 0, 0)), ('SND', (0, 1, 0, 1)), ('XOR', (0, 1, 1, 0)), ('LOR', (0, 1, 1, 1)), ('NOR', (1, 0, 0, 0)), ('XNR', (1, 0, 0, 1)), ('NSD', (1, 0, 1, 0)), ('LIF', (1, 0, 1, 1)), ('NFT', (1, 1, 0, 0)), ('IMP', (1, 1, 0, 1)), ('NND', (1, 1, 1, 0)), ('TRU', (1, 1, 1, 1))]

List of pairs of string representations and corresponding unary/binary operations.

static parse(*token: str*) → *bfcl.bfcl.operation*

Parse a Bristol Fashion circuit gate operator token.

```
>>> operation.parse('AND')
(0, 0, 0, 1)
```

emit() → str

Emit a Bristol Fashion operation token.

```
>>> operation((0, 1, 1, 0)).emit()
'XOR'
```

bfcl.bfcl.**op**

alias of `bfcl.bfcl.operation`

class bfcl.bfcl.**gate**(*wire_in_count: Optional[int] = None*, *wire_out_count: Optional[int] = None*, *wire_in_index: Optional[Sequence[int]] = None*, *wire_out_index: Optional[Sequence[int]] = None*, *operation: Optional[bfcl.bfcl.operation] = None*)

Bases: `object`

Data structure for an individual circuit logic gate.

```
>>> gate.parse('2 1 0 1 15 AND').emit()
'2 1 0 1 15 AND'
>>> gate.parse('1 1 100 200 INV').emit()
'1 1 100 200 INV'
```

static parse(*tokens*) → *bfcl.bfcl.gate*
Parse a Bristol Fashion gate string or token list.

emit() → str
Emit a Bristol Fashion string for this gate.

class bfcl.bfcl.**bfc**(*raw=None*)
Bases: `object`

Data structure for circuits represented using the Bristol Fashion. A string representing a circuit that conforms to the Bristol Fashion syntax can be parsed into an instance of this class.

```
>>> circuit_string = ['7 36', '2 4 4', '1 1']
>>> circuit_string.extend(['2 1 0 1 15 AND', '2 1 2 3 16 AND'])
>>> circuit_string.extend(['2 1 15 16 8 AND', '2 1 4 5 22 AND'])
>>> circuit_string.extend(['2 1 6 7 23 AND', '2 1 22 23 9 AND'])
>>> circuit_string.extend(['2 1 8 9 35 AND'])
>>> circuit_string = "\n".join(circuit_string)
>>> c = bfc(circuit_string)
```

The string representation can be recovered from an instance of this class, as well.

```
>>> c.emit() == circuit_string
True
>>> for line in c.emit().split("\n"):
...     print(line)
7 36
2 4 4
1 1
2 1 0 1 15 AND
2 1 2 3 16 AND
2 1 15 16 8 AND
2 1 4 5 22 AND
2 1 6 7 23 AND
2 1 22 23 9 AND
2 1 8 9 35 AND
```

We could just add a '1 1 35 36 LID' line, and increment '8 16', but the *force_id_outputs* is perhaps not as lazy as it could be and performs a full *bfc*->`circuit_`->`bfc` conversion to get the identity gates, hence the wire renumbering. >>> for line in c.emit(force_id_outputs=True).split("n"): … print(line) 8 16 2 4 4 1 1 2 1 0 1 8 AND 2 1 2 3 9 AND 2 1 8 9 10 AND 2 1 4 5 11 AND 2 1 6 7 12 AND 2 1 11 12 13 AND 2 1 10 13 14 AND 1 1 14 15 LID

A circuit can also be consructed using an instance of the `circuit` class defined in the circuit library (see the documentation for the `circuit.circuit` method defined as part of this class).

Common properties of the circuit can be found in the attributes of an instance.

```
>>> c.gate_count
7
>>> c.wire_count
```

(continues on next page)

(continued from previous page)

```
36
>>> c.value_in_count
2
>>> c.value_in_length
[4, 4]
>>> c.value_out_count
1
>>> c.wire_in_count
8
>>> c.wire_in_index
[0, 1, 2, 3, 4, 5, 6, 7]
>>> c.wire_out_count
1
>>> c.wire_out_index
[35]
```

The individual gates are stored within a list consisting of zero or more instances of the `gate` class.

```
>>> (c.gate[0].wire_in_index, c.gate[0].wire_out_index)
([0, 1], [15])
>>> (c.gate[1].wire_in_index, c.gate[1].wire_out_index)
([2, 3], [16])
>>> (c.gate[2].wire_in_index, c.gate[2].wire_out_index)
([15, 16], [8])
>>> (c.gate[3].wire_in_index, c.gate[3].wire_out_index)
([4, 5], [22])
>>> (c.gate[4].wire_in_index, c.gate[4].wire_out_index)
([6, 7], [23])
>>> (c.gate[5].wire_in_index, c.gate[5].wire_out_index)
([22, 23], [9])
>>> (c.gate[6].wire_in_index, c.gate[6].wire_out_index)
([8, 9], [35])
>>> {c.gate[i].operation for i in range(7)} == {op.and_}
True
```

A circuit can also be evaluated an on a sequence of input bit vectors using the `bfcl.evaluate` method.

```
>>> from itertools import product
>>> inputs = list(product(*([[0, 1]]*4)))
>>> pairs = product(inputs, inputs)
>>> outputs = ([0]*255) + [1]
>>> [c.evaluate(p)[0][0] for p in pairs] == outputs
True
```

circuit(*c: Optional[circuit.circuit.circuit] = None*) → Union[Type[None], circuit.circuit.circuit]

Populate this Bristol Fashion circuit instance using an instance of the `circuit` class defined in the circuit library.

```
>>> c_ = circuit_.circuit()
>>> c_.count()
0
>>> g0 = c_.gate(op.id_, is_input=True)
>>> g1 = c_.gate(op.id_, is_input=True)
```

(continues on next page)

(continued from previous page)

```
>>> g2 = c_.gate(op.and_, [g0, g1])
>>> g3 = c_.gate(op.id_, [g2], is_output=True)
>>> c_.count()
4
>>> c = bfc(c_)
>>> c.emit().split("\n")
['2 4', '1 2', '1 1', '2 1 0 1 2 AND', '1 1 2 3 LID']
>>> c_reparsed = bfc(bfc(c_).circuit())
>>> c_reparsed.emit().split("\n")
['2 4', '1 2', '1 1', '2 1 0 1 2 AND', '1 1 2 3 LID']
```

parse(*raw: str*)

Parse a string representation of a circuit that conforms to the Bristol Fashion syntax.

```
>>> s = ['7 36', '2 4 4', '1 1']
>>> s.extend(['2 1 0 1 15 AND', '2 1 2 3 16 AND'])
>>> s.extend(['2 1 15 16 8 AND', '2 1 4 5 22 AND'])
>>> s.extend(['2 1 6 7 23 AND', '2 1 22 23 9 AND'])
>>> s.extend(['2 1 8 9 35 AND'])
>>> s = "\n".join(s)
>>> c = bfc()
>>> c.parse(s)
>>> for line in c.emit().split("\n"):
...     print(line)
7 36
2 4 4
1 1
2 1 0 1 15 AND
2 1 2 3 16 AND
2 1 15 16 8 AND
2 1 4 5 22 AND
2 1 6 7 23 AND
2 1 22 23 9 AND
2 1 8 9 35 AND
```

emit(*force_id_outputs=False*, *progress=lambda _: ...*) → str

Emit a string representation of a Bristol Fashion circuit definition.

In the example below, a circuit object is first constructed using the circuit library.

```
>>> c_ = circuit_.circuit()
>>> c_.count()
0
>>> g0 = c_.gate(op.id_, is_input=True)
>>> g1 = c_.gate(op.id_, is_input=True)
>>> g2 = c_.gate(op.and_, [g0, g1])
>>> g3 = c_.gate(op.id_, [g2], is_output=True)
```

The `c_` object above can be converted into an instance of the class `circuit`.

```
>>> c = bfc(c_)
```

This method can be used to emit a string representation of an object, where the string conforms to the Bristol Fashion syntax.

```
>>> c.emit().split("\n")
['2 4', '1 2', '1 1', '2 1 0 1 2 AND', '1 1 2 3 LID']
```

```
>>> c.emit(True).split("\n")
['2 4', '1 2', '1 1', '2 1 0 1 2 AND', '1 1 2 3 LID']
```

evaluate(*inputs: Sequence[Sequence[int]]*) → Sequence[Sequence[int]]

Evaluate a circuit on a sequence of input bit vectors.

```
>>> s = ['7 36', '2 4 4', '1 1']
>>> s.extend(['2 1 0 1 15 AND', '2 1 2 3 16 AND'])
>>> s.extend(['2 1 15 16 8 AND', '2 1 4 5 22 AND'])
>>> s.extend(['2 1 6 7 23 AND', '2 1 22 23 9 AND'])
>>> s.extend(['2 1 8 9 35 AND'])
>>> c = bfc("\n".join(s))
>>> c.evaluate([[1, 0, 1, 1], [1, 1, 1, 0]])
[[0]]
>>> c.evaluate([[1, 1, 1, 1], [1, 1, 1, 1]])
[[1]]
```

The example below confirms that the circuit `c` defined above has correct behavior when evaluated on all compatible inputs (*i.e.*, inputs consisting of a pair of 4-bit vectors).

```
>>> from itertools import product
>>> inputs = list(product(*([[0, 1]]*4)))
>>> pairs = product(inputs, inputs)
>>> outputs = ([0]*255) + [1]
>>> [c.evaluate(p)[0][0] for p in pairs] == outputs
True
```

The documentation can be generated automatically from the source files using Sphinx:

```
cd docs
python -m pip install -r requirements.txt
sphinx-apidoc -f -E --templatedir=_templates -o _source .. ../setup.py && make html
```

CHAPTER FOUR

TESTING AND CONVENTIONS

All unit tests are executed and their coverage is measured when using pytest (see `setup.cfg` for configuration details):

```
python -m pip install pytest pytest-cov
python -m pytest
```

Alternatively, all unit tests are included in the module itself and can be executed using doctest:

```
python bfcl/bfcl.py -v
```

Style conventions are enforced using Pylint:

```
python -m pip install pylint
python -m pylint bfcl
```

CHAPTER

FIVE

CONTRIBUTIONS

In order to contribute to the source code, open an issue or submit a pull request on the GitHub page for this library.

CHAPTER

SIX

VERSIONING

Beginning with version 0.2.0, the version number format for this library and the changes to the library associated with version number increments conform with Semantic Versioning 2.0.0.

CHAPTER SEVEN

PUBLISHING

This library can be published as a package on PyPI by a package maintainer. Install the wheel package, remove any old build/distribution files, and package the source into a distribution archive:

```
python -m pip install wheel
rm -rf dist *.egg-info
python setup.py sdist bdist_wheel
```

Next, install the twine package and upload the package distribution archive to PyPI:

```
python -m pip install twine
python -m twine upload dist/*
```

PYTHON MODULE INDEX

b

INDEX