`galactic` practice guide

The Galactic Organization <contact@thegalactic.org>

0.0.8

Contents

List of Figures

1 Introduction

This document is produced under the CC-by-nc-nd licence [1]

The *galactic* project is architecturally designed (fig. 1) with a core library and a set of plugins to increase its functionalities.

[1] © 2018-2019 the Galactic Organization. This document is licensed under CC-by-nc-nd (https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en)

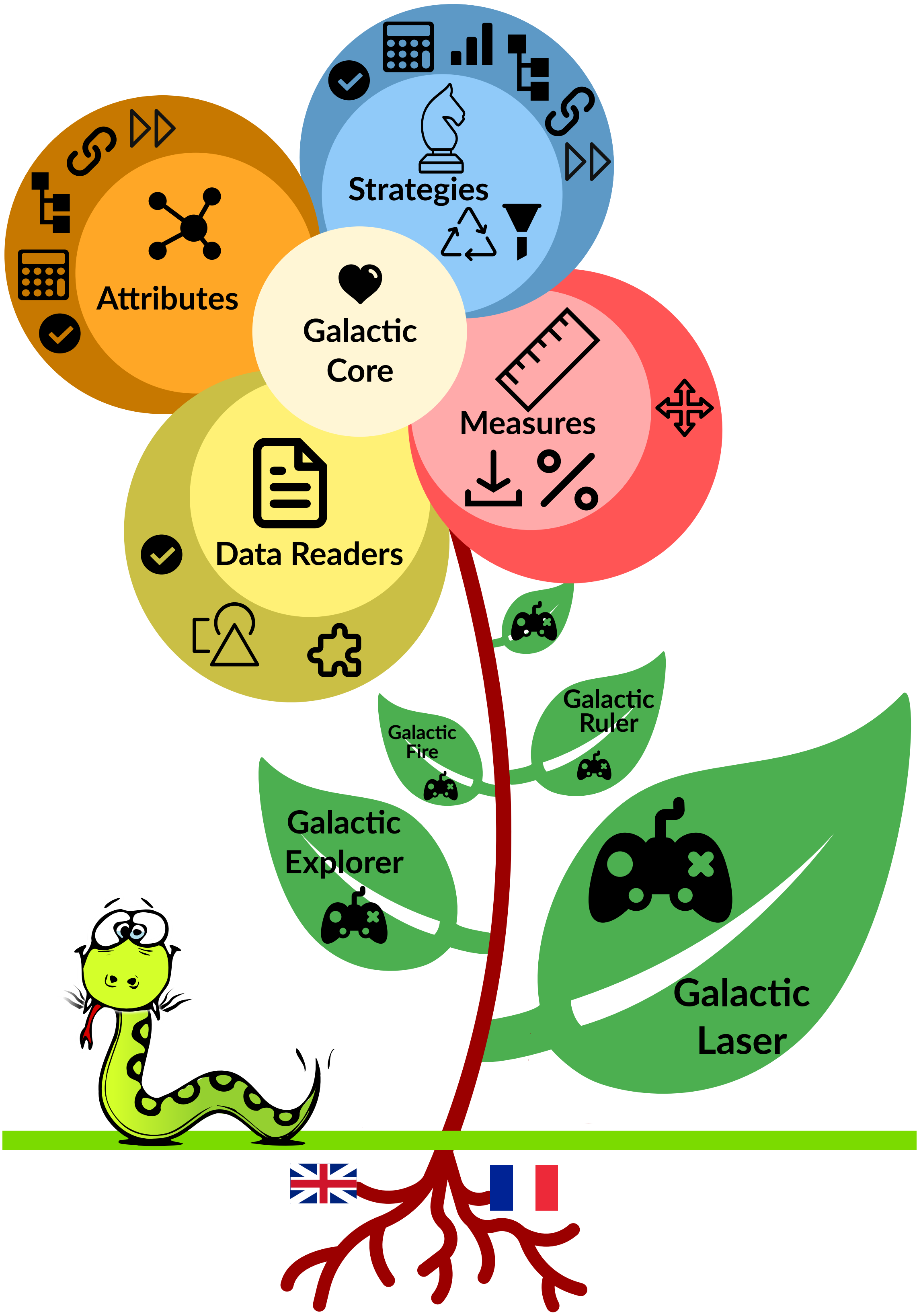


Figure 1: *Galactic* architecture

This practical guide is a collection of all the jupyter notebooks present in the core and in the plugins.

All lines

```python
print("test")
```

are python input.

And all lines

```
test
```

are python output.

2 *core* library

2.1 Algebras

2.1.1 Partially ordered set

2.1.1.1 Elements

Partially ordered elements implements the 6 classical comparison operation:

- $<$
- $\leq$
- $>$
- $\geq$
- $=$
- $\neq$

They can be filtered using some special functions.

Integer example The `Integer` class in the examples folder implements a partially ordered relation between positive integers using the *divisor of* notion.

To import the `Integer` class:

```python
from galactic.examples.arithmetic.algebras import Integer
Integer(12)
```

$12 = 2^2 3^1$

3 is lesser than 6:

```
Integer(3) <= Integer(6)
```

```
True
```

2 and 3 are incomparable:

```
Integer(2) <= Integer(3)
```

```
False
```

```
Integer(3) <= Integer(2)
```

```
False
```

1 is lesser than all the other numbers (1 is a divisor of all numbers) and 0 is greater than all other numbers (all numbers are divisors of 0):

```
Integer(1) <= Integer(6)
```

```
True
```

```
Integer(6) <= Integer(0)
```

```
True
```

```
from galactic.algebras.poset.elements import top
elements = [Integer(6), Integer(24), Integer(13)]
top(elements)
```

```
<odict_iterator at 0x7ff6c45a5b48>
```

Note that the result is a python iterator. To get the list:

```
display(*top(elements))
```

$24 = 2^3 3^1$

$13 = 13^1$

An analog operation is available to get the *bottom* elements:

```
from galactic.algebras.poset.elements import bottom
```

```
display(*bottom(elements))
```

$6 = 2^1 3^1$

$13 = 13^1$

It’s possible to get the elements greater or lower than a given limit:

```
from galactic.algebras.poset.elements import upper_limit
display(*upper_limit(iterable=elements, limit=Integer(6)))
```

$6 = 2^1 3^1$

```
from galactic.algebras.poset.elements import lower_limit
display(*lower_limit(iterable=elements, limit=Integer(12), strict=True))
```

$24 = 2^3 3^1$

Partially ordered elements can be lower (and upper) bounded.

This is the case of the `Integer` class:

```
Integer.minimum()
```

1

```
Integer.maximum()
```

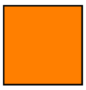
0

Color example The `Color` class in the examples folder implements a partially ordered relation between colors.

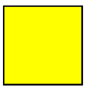
To import the `Color` class:

```
from galactic.examples.color.algebras import Color
```

```
Color(red=1.0, green=0.5)
```



```
Color(red=1.0, green=1.0)
```

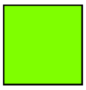


The orange color is less or equal than the yellow color:

```
Color(red=1.0, green=0.5) <= Color(red=1.0, green=1.0)
```

```
True
```

```
Color(red=0.5, green=1.0)
```



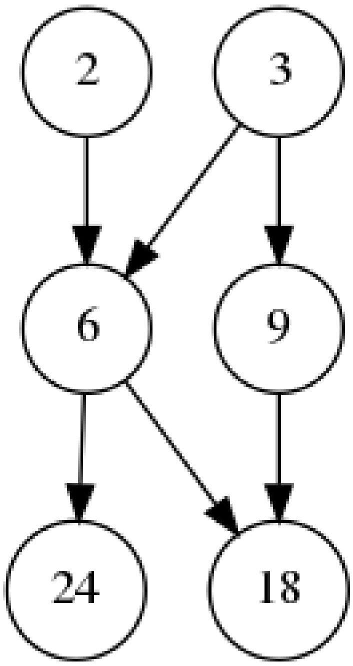
2.1.1.2 Collections A *poset* is a set or partially ordered elements. You can use either the `BasicPartiallyOrderedSet` class or the `CompactPartiallyOrderedSet` class.

Integer example

```
from galactic.algebras.poset.collections import BasicPartiallyOrderedSet
poset = BasicPartiallyOrderedSet([
    Integer(24),
    Integer(18),
    Integer(9),
    Integer(6),
    Integer(3),
    Integer(2)
])
```

poset have all classical python operations on sets:

```
poset
```




```
len(poset)
```

```
6
```

```
display(*poset)
```

$2 = 2^1$
$3 = 3^1$
$6 = 2^1 3^1$
$9 = 3^2$
$18 = 2^1 3^2$
$24 = 2^3 3^1$

```
Integer(3) in poset
```

```
True
```

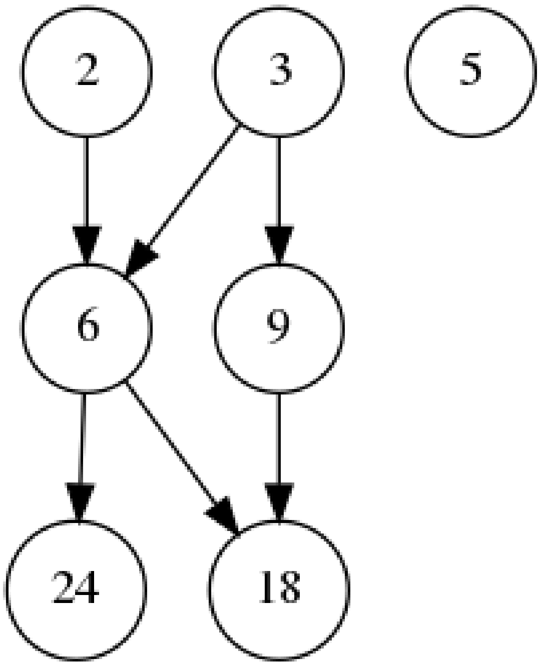
```
display(*(poset & BasicPartiallyOrderedSet([Integer(9), Integer(6), Integer(5)]
```

$9 = 3^2$
$6 = 2^1 3^1$

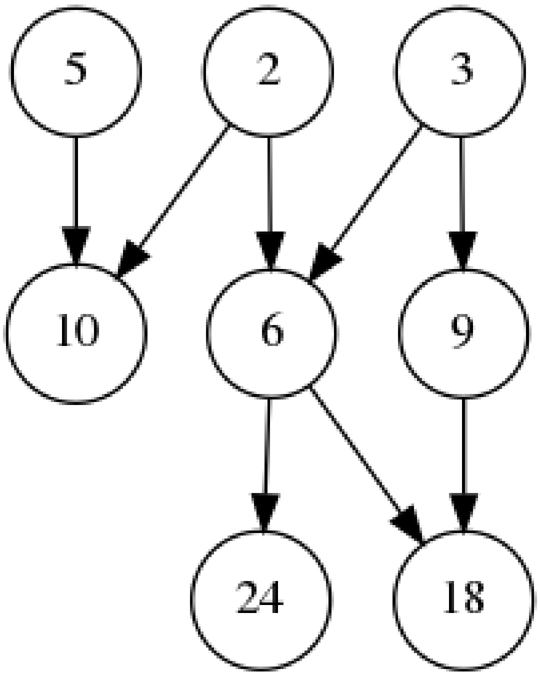
```
display(*(poset | BasicPartiallyOrderedSet([Integer(9), Integer(6), Integer(5)]
```

$2 = 2^1$
$3 = 3^1$
$18 = 2^1 3^2$
$5 = 5^1$
$6 = 2^1 3^1$
$24 = 2^3 3^1$
$9 = 3^2$

```python
poset @= BasicPartiallyOrderedSet([Integer(9), Integer(6), Integer(5)])
poset
```




```python
poset += [Integer(10)]
poset
```




```
poset >= BasicPartiallyOrderedSet([Integer(9), Integer(6)])
```

```
True
```

From a *poset*, several additional methods can be applied:

- get the *top* or the *bottom* elements from the *poset*;
- get the *descendants* or the *ascendants* of an element;
- get the *successors* and *predecessors* of an element;
- get the *siblings*, *co-parents* or *neighbors* (*sibling* or *co-parent*) of an element.

```
display(*poset.top())
```

$10 = 2^1 5^1$
$18 = 2^1 3^2$
$24 = 2^3 3^1$

```
display(*poset.bottom())
```

$2 = 2^1$
$3 = 3^1$
$5 = 5^1$

```
display(*poset.descendants(Integer(18)))
```

```
display(*poset.ascendants(Integer(6)))
```

$2 = 2^1$
$3 = 3^1$

```
display(*poset.successors(Integer(6)))
```

$24 = 2^3 3^1$
$18 = 2^1 3^2$

```
display(*poset.predecessors(Integer(6)))
```

$2 = 2^1$
$3 = 3^1$

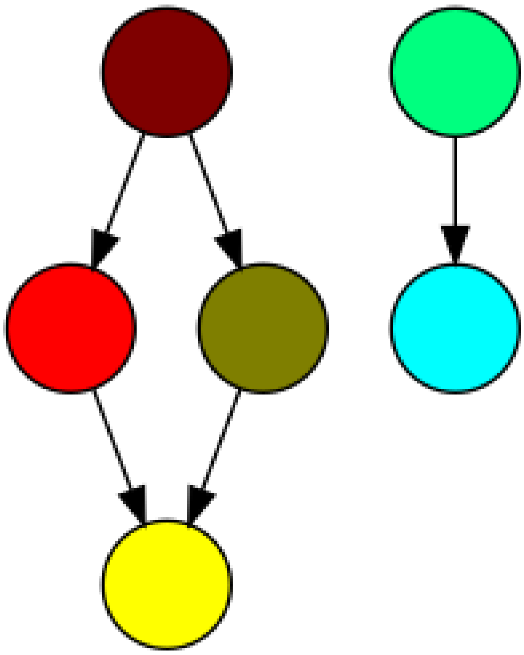
```
display(*poset.siblings(Integer(6)))
```

$9 = 3^2$
$10 = 2^1 5^1$

Color example Using colors, you can can test the poset collections:

```
from galactic.algebras.poset.collections import CompactPartiallyOrderedSet
poset = CompactPartiallyOrderedSet([
    Color(red=1.0),
    Color(red=1.0, green=1.0),
    Color(red=0.5),
    Color(red=0.5, green=0.5),
    Color(green=1.0, blue=0.5),
    Color(green=1.0, blue=1.0)
])
```

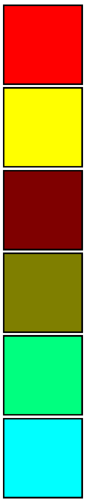
```
poset
```



```
len(poset)
```

```
6
```

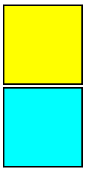
```
display(*poset)
```



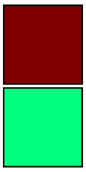
```
poset >= CompactPartiallyOrderedSet([
    Color(green=1.0, blue=0.5),
    Color(green=1.0, blue=1.0)
])
```

```
True
```

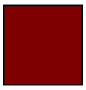
```python
display(*poset.top())
```



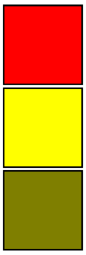
```python
display(*poset.bottom())
```



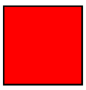
```python
Color(red=0.5)
```



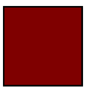
```python
display(*poset.descendants(Color(red=0.5)))
```



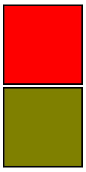
```python
Color(red=1.0)
```



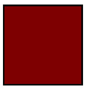
```python
display(*poset.ascendants(Color(red=1.0)))
```



```python
display(*poset.successors(Color(red=0.5)))
```



```
display(*poset.predecessors(Color(red=1.0)))
```



2.1.2 Lattice

2.1.2.1 Elements Lattice are a particular case of *poset*. Their elements have the following properties:

For all x, y of a poset, there exists a unique infimum and a unique supremum:

Integer example

```
from galactic.examples.arithmetic.algebras import Integer
```

```
Integer(24) & Integer(18)
```

$6 = 2^1 3^1$

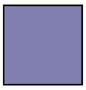
```
Integer(24) | Integer(18)
```

$72 = 2^3 3^2$

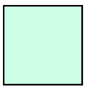
Color example

```
from galactic.examples.color.algebras import Color
```

```
Color(red=0.8, green=0.5, blue=0.7) & Color(red=0.5, green=1.0, blue=0.9)
```



```
Color(red=0.8, green=0.5, blue=0.7) | Color(red=0.5, green=1.0, blue=0.9)
```



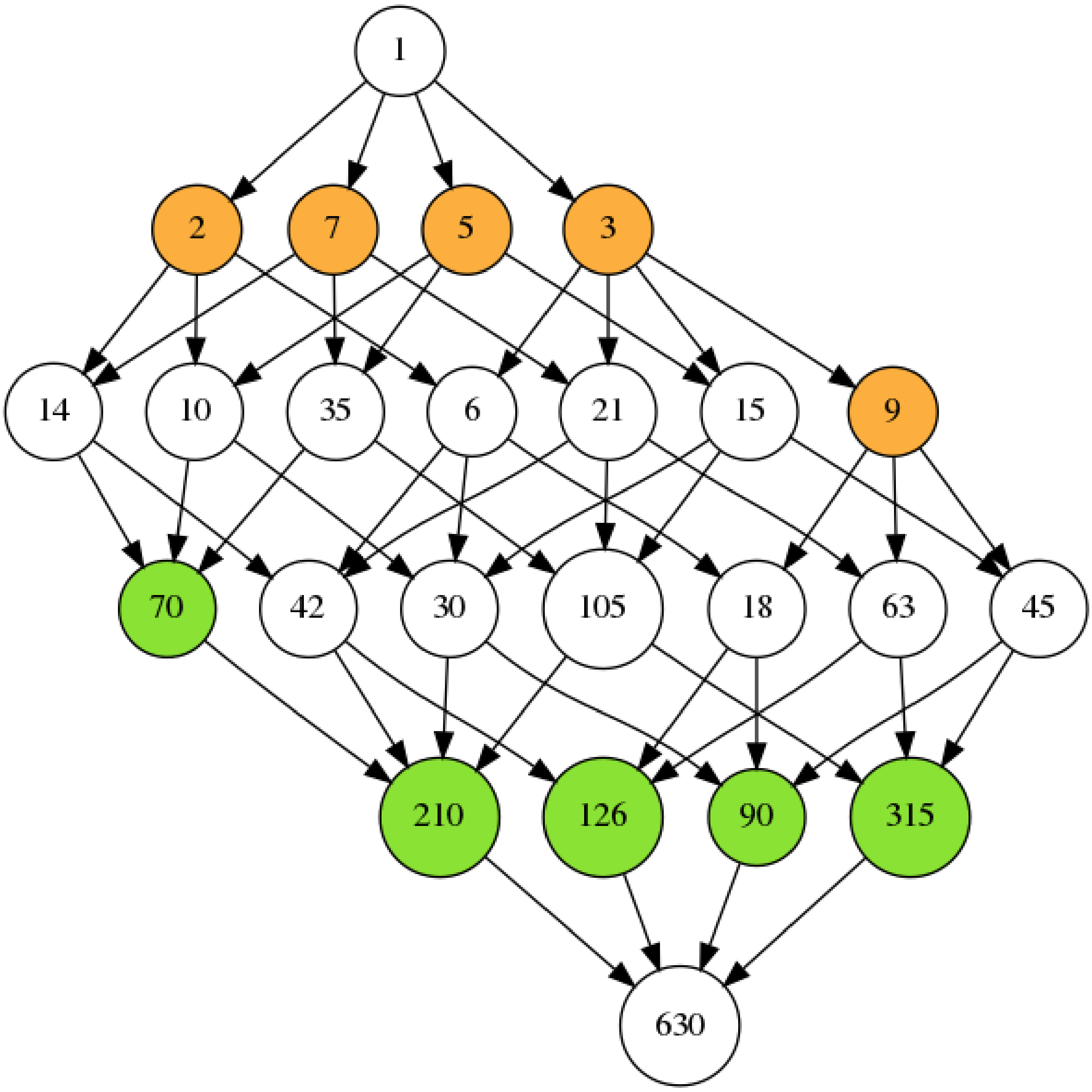
2.1.2.2 Collections All operations on *poset* are available and some extras features:

- get the *minimum* and *maximum* elements from a lattice;
- get the *atoms* and the *co-atoms* elements from a lattice;
- get the *meet-irreducible* and the *join-irreducible* elements from a lattice;
- get the *meet-irreducible generators* and the *join-irreducible generators* of an element of the lattice.

Integer example

```
from galactic.algebras.lattice.collections import BasicLattice
```

```
integers = [Integer(2), Integer(3), Integer(5), Integer(7), Integer(9)]
lattice = BasicLattice(integers)
lattice
```




```
display(*lattice)
```

$2 = 2^1$
$3 = 3^1$
$5 = 5^1$
$6 = 2^1 3^1$
$7 = 7^1$
$9 = 3^2$
$10 = 2^1 5^1$
$42 = 2^1 3^1 7^1$
$45 = 3^2 5^1$
$14 = 2^1 7^1$
$15 = 3^1 5^1$
$18 = 2^1 3^2$
$21 = 3^1 7^1$
$30 = 2^1 3^1 5^1$
$35 = 5^1 7^1$
$315 = 3^2 5^1 7^1$
$63 = 3^2 7^1$
$70 = 2^1 5^1 7^1$
$210 = 2^1 3^1 5^1 7^1$
$90 = 2^1 3^2 5^1$
$105 = 3^1 5^1 7^1$
$630 = 2^1 3^2 5^1 7^1$
$126 = 2^1 3^2 7^1$
1

```
lattice.minimum()
```

1

```
lattice.maximum()
```

$630 = 2^1 3^2 5^1 7^1$

```
display(*lattice.atoms())
```

$2 = 2^1$
$3 = 3^1$
$5 = 5^1$
$7 = 7^1$

```
display(*lattice.co_atoms())
```

$315 = 3^2 5^1 7^1$
$210 = 2^1 3^1 5^1 7^1$
$90 = 2^1 3^2 5^1$
$126 = 2^1 3^2 7^1$

```python
display(*lattice.meet_irreducible())
```

$315 = 3^2 5^1 7^1$
$70 = 2^1 5^1 7^1$
$210 = 2^1 3^1 5^1 7^1$
$90 = 2^1 3^2 5^1$
$126 = 2^1 3^2 7^1$

```python
display(*lattice.join_irreducible())
```

$2 = 2^1$
$3 = 3^1$
$5 = 5^1$
$7 = 7^1$
$9 = 3^2$

```python
display(*lattice.smallest_meet_irreducible(Integer(30)))
```

$210 = 2^1 3^1 5^1 7^1$
$90 = 2^1 3^2 5^1$

```python
display(*lattice.greatest_join_irreducible(Integer(30)))
```

$2 = 2^1$
$3 = 3^1$
$5 = 5^1$

The context can be saved in yaml format as an input to formal concept analysis.

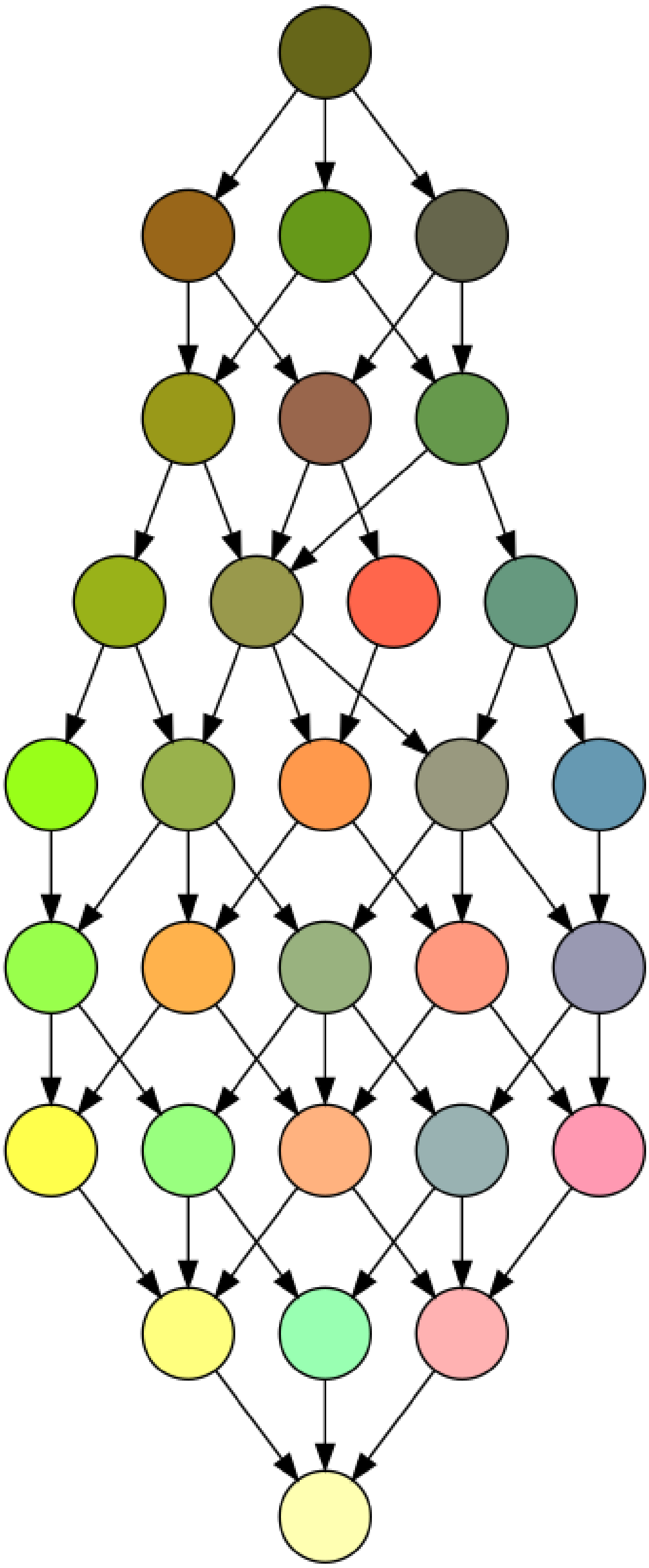
Color example

```python
from galactic.examples.color.algebras import Color
from galactic.algebras.lattice.collections import CompactLattice
```

```python
lattice = CompactLattice([
    Color(red=1.0, green=0.7, blue=0.5),
    Color(red=0.6, green=1.0, blue=0.1),
```

```
    Color(red=0.4, green=0.6, blue=0.7),
    Color(red=1.0, green=0.4, blue=0.3),
])
```

```
lattice
```



2.1.2.3 Context There exists a bijection between a lattice and its minimal binary table called its equivalent context:

- the rows are composed by the *join-irreducible*
- the columns are composed by the *meet-irreducible*
- the boolean value for row i and column j is True if $i \leq j$

Integer example

```
integers = [Integer(2), Integer(3), Integer(5), Integer(7), Integer(9)]
lattice = BasicLattice(integers)
context = {
    str(irreducible1): [
        str(irreducible2)
        for irreducible2 in lattice.meet_irreducible()
        if irreducible1 <= irreducible2
    ]
    for irreducible1 in lattice.join_irreducible()
}
context
```

```
{'2': ['70', '210', '90', '126'],
 '3': ['315', '210', '90', '126'],
 '5': ['315', '70', '210', '90'],
 '7': ['315', '70', '210', '126'],
 '9': ['315', '90', '126']}
```

```
import tempfile
import yaml
with tempfile.TemporaryFile(mode="w+t") as file:
    yaml.dump(context, file)
    file.seek(0)
    print(file.read())
```

```
'2':
- '70'
- '210'
- '90'
- '126'
```

```
'3':
- '315'
- '210'
- '90'
- '126'
'5':
- '315'
- '70'
- '210'
- '90'
'7':
- '315'
- '70'
- '210'
- '126'
'9':
- '315'
- '90'
- '126'
```

2.2 Concepts

2.2.1 Concept lattice

A population can be created using a collection of python objects:

```python
from galactic.concepts import Population
individuals = {48: 48, 36: 36, 64: 64, 56:56, 84: 84}
population=Population(individuals)
population
```

```
<galactic.concepts.Population at 0x7fa98c3f72e8>
```

```python
list(population)
```

```
['48', '36', '64', '56', '84']
```

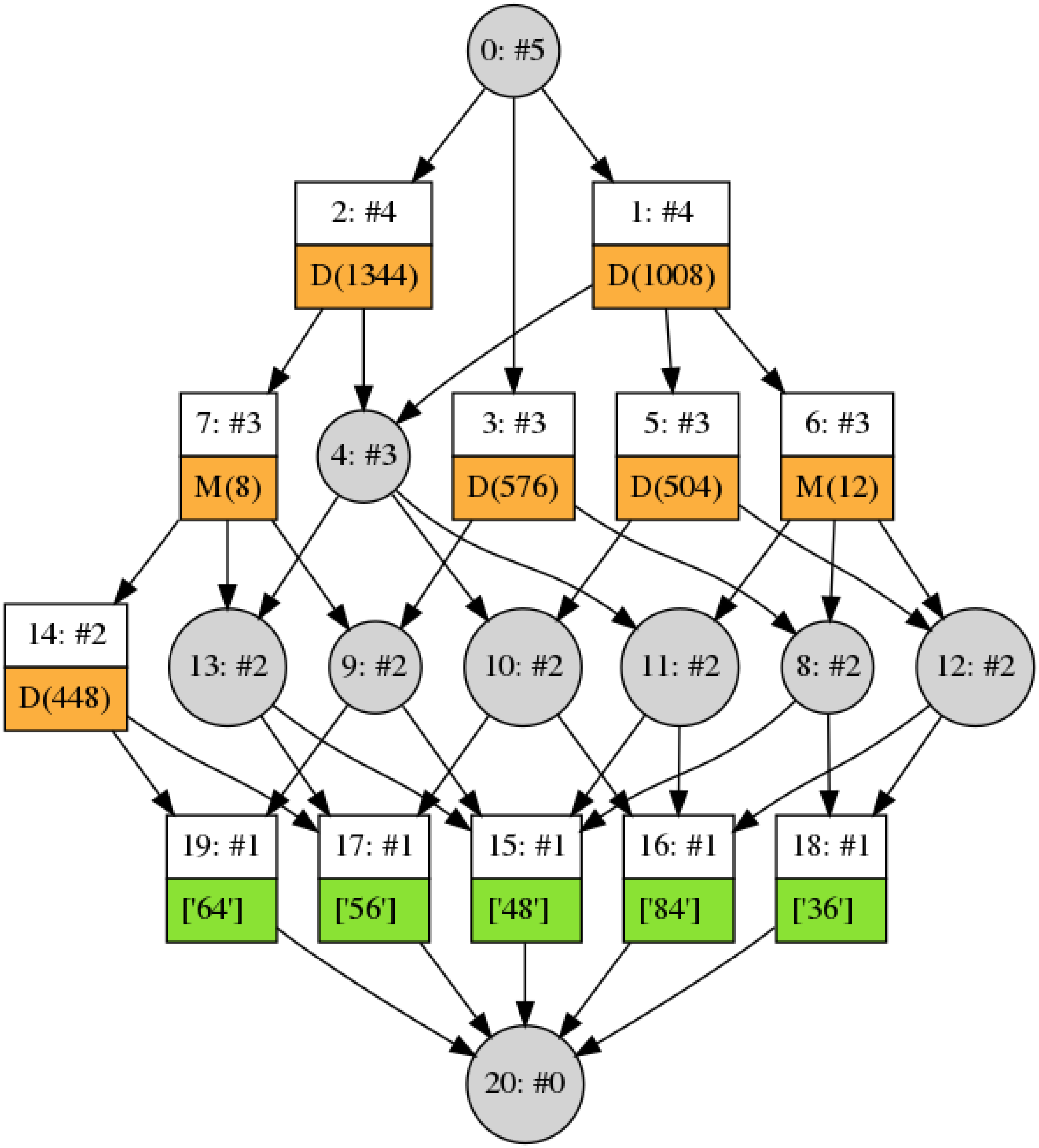
A lattice can be created from a population using a set of strategies:

```python
from galactic.concepts import Lattice
from galactic.attributes import identity
from galactic.examples.arithmetic.strategies import IntegerStrategy
lattice = Lattice(population=population, strategies=[IntegerStrategy(identity)]
```

```python
list(str(concept) for concept in lattice)
```

```
['M(4) and D(4032)',
 'M(4) and D(1008)',
 'M(4) and D(1344)',
 'M(4) and D(576)',
 'M(4) and D(336)',
 'M(4) and D(504)',
 'M(12) and D(1008)',
 'M(8) and D(1344)',
 'M(12) and D(144)',
 'M(16) and D(192)',
 'M(28) and D(168)',
 'M(12) and D(336)',
 'M(12) and D(252)',
 'M(8) and D(336)',
 'M(8) and D(448)',
 'M(48) and D(48)',
 'M(84) and D(84)',
 'M(56) and D(56)',
 'M(36) and D(36)',
 'M(64) and D(64)',
 'False']
```

```python
lattice
```



3 Plugins

3.1 Attributes

3.1.1 Logical attributes

3.1.1.1 Predicates The *py-attribute-logical* plugin defines several logical predicates.

***not* predicate**

```
from galactic.attributes import Key
from galactic_attribute_logical import Not
a = Not(Key(name="x"))
a
```

$\neg x$

a applied to an individual whose key paremeter x is `True` gives `False`

```
a({"x": True})
```

```
False
```

a applied to an individual without key parameter x gives `True`

```
a({})
```

```
True
```

***and* predicate**

```
from galactic.attributes import Key
from galactic_attribute_logical import And
a = And(Key(name="x"), Key(name="y"))
a
```

$(x \wedge y)$

a applied to an individual whose key parameters x and y are `True` gives `True`

```
a({"x": True, "y": True})
```

```
True
```

a applied to an individual without key parameter y gives `False`

```
a({"x": True})
```

```
False
```

a applied to a None individual gives `False`

```python
a()
```

```
False
```

An *and* constructed without inner attributes is always `True` (`True` is the neutral element for the *and* operation)

```python
And()
```

$\top$

```python
And()()
```

```
True
```

***or* predicates**

```python
from galactic.attributes import Key
from galactic_attribute_logical import Or
a = Or(Key(name="x"), Key(name="y"))
a
```

$(x \vee y)$

a applied to an individual whose key parameters x and y are `True` gives `True`

```python
a({"x": True, "y": True})
```

```
True
```

a applied to an individual whose key parameter x is `True` gives `True`

```python
a({"x": True})
```

```
True
```

a applied to an individual whose key parameter y is `True` gives `True`

```python
a({"y": True})
```

```
True
```

a applied to an empty individual gives True

```python
a({})
```

```
False
```

a applied to an None individual gives True

```python
a()
```

```
False
```

An *or* constructed without inner attributes is always False (False is the neutral element for the *or* operation)

```python
Or()
```

⊥

```python
Or()()
```

```
False
```

xor predicate

```python
from galactic.attributes import Key
from galactic_attribute_logical import Xor
a = Xor(Key(name="x"), Key(name="y"))
a
```

$(x \oplus y)$

a applied to an individual whose key parameter x is True and without key parameter y gives True

```python
a({"x": True})
```

```
True
```

a applied to an individual whose key parameters x and y are True gives False

```python
a({"x": True, "y": True})
```

```
False
```

a applied to an empty individual gives False

```python
a({})
```

```
False
```

a applied to an None individual gives False

An *xor* constructed without inner attributes is always False

```python
Xor()
```

⊥

```python
Xor()()
```

```
False
```

***equivalence* predicate**

```python
from galactic.attributes import Key
from galactic_attribute_logical import Equivalence
a = Equivalence(Key(name="x"), Key(name="y"))
a
```

$(x \equiv y)$

a applied to an individual whose key parameter x is True and without key parameter y gives False

```python
a({"x": True})
```

```
False
```

a applied to an individual whose key parameters x and y are True gives True

```python
a({"x": True, "y": True})
```

```
True
```

a applied to an empty individual gives `True`

```python
a({})
```

```
True
```

An *equivalence* constructed without inner attributes is always `True`

```python
Equivalence()
```

$\top$

```python
Equivalence()()
```

```
True
```

3.1.1.2 Descriptions

Boolean description The `BooleanDescription` class is used to represent classical boolean attributes in formal concept analysis.

```python
from galactic_attribute_logical import BooleanDescription
from galactic.attributes import Key
description = BooleanDescription(Key(name="x"))
```

The description applied to a list of two individuals whose key parameters x is equal to `True` gives the singleton x

```python
display(*(attribute for attribute in description([{"x": 1}, {"x": True}])))
```

x

The description applied to a list of two individuals whose key parameters x is equal to `True` and `False` gives an empty set.

```python
display(*(attribute for attribute in description([{"x": True}, {"x": False}])))
```

The description applied to an empty list gives the singleton x (all individuals have the attribute x)

```
display(*(attribute for attribute in description([])))
```

x

Logical description The LogicalDescription class is used to represent logical description spaces.

It computes the description of a collection of individuals using the Quine-McCluskey algorithm on the complement of individuals to obtain clauses. It is not designed to be ran on large number of boolean attributes since the complexity is $O(3^n \log(n))$ (n being the number of boolean variables).

```
from galactic_attribute_logical import LogicalDescription
from galactic.attributes import Key
description = LogicalDescription(Key(name="x"), Key(name="y"), Key(name="z"))
```

```
display(*(attribute for attribute in description([{"x": 0, "y": 1}, {"x": 1, "y
```

(x ∨ y)
¬z
(¬x ∨ ¬y)

```
display(*(attribute for attribute in description([])))
```

⊥

3.1.2 Numerical attributes

3.1.2.1 Attributes The *py-attribute-numerical* plugin defines several numerical attributes:

***Number* attribute** The Number attribute can convert any value to a numerical value

```
from galactic.attributes import Key
from galactic_attribute_numerical import Number
a = Number(Key(name="x"))
print(a)
```

x

```
a({"x": 5})
```

```
5.0
```

```
a(4)
```

```
nan
```

***Linear* attribute** The `Linear` attribute can do a linear transformation of a numerical value

```
from galactic_attribute_numerical import Linear
l = Linear(Key(name="x"), coefficient=2)
print(l)
```

```
2*x
```

```
l({"x": 5})
```

```
10.0
```

```
l(None)
```

```
nan
```

3.1.2.2 Predicates

***Positive* predicate** The `Positive` predicate can test if an individual has a positive value for a numerical attribute.

```
from galactic_attribute_numerical import upper_limit
u = upper_limit(Key(name="x"), limit=2)
print(u)
```

```
x<=2
```

```
u({})
```

```
False
```

```python
u({"x": 0})
```

```
True
```

***upper_limit* and *lower_limit* functions** The upper_limit and lower_limit are convenient function to create Positive attributes.

```python
from galactic_attribute_numerical import lower_limit
l = lower_limit(Key(name="x"), limit=2)
print(l)
```

```
x>=2.0
```

```python
l({})
```

```
False
```

```python
l({"x": 2})
```

```
True
```

```python
from galactic_attribute_numerical import upper_limit
u = upper_limit(Key(name="x"), limit=2)
print(u)
```

```
x<=2
```

```python
u({})
```

```
False
```

```python
u({"x": 0})
```

```
True
```

3.1.2.3 Descriptions

NumericalDescription A `NumericalDescription` can describe a collection of individuals by calculating the convex hull of numerical attributes.

If the number of numerical attributes is equal to

- 1: it produces intervals on the real line;
- 2: it produces polygons on R^2

```
from galactic_attribute_numerical import NumericalDescription
description = NumericalDescription(Key(name="x"))
predicates = description([{"x": 0}, {"x": 1}, {"x": 2}])
[str(predicate) for predicate in predicates]
```

```
['x>=0.0', 'x<=2']
```

```
predicates = description([{"x": 0}, {"x": 1}, {}])
[str(predicate) for predicate in predicates]
```

```
[]
```

3.1.3 Categorized attributes

3.1.3.1 Predicates The *py-attribute-categorized* plugin defines several categorized predicates:

***CategoryPredicate* attribute** The `CategoryPredicate` attribute can affirm if a value is in a defined set.

```
from galactic.attributes import Key
from galactic_attribute_categorized import CategoryPredicate
a = CategoryPredicate(Key(name="x"), values=frozenset({1, 2}))
print(a)
```

```
x in {1, 2}
```

```
a.values
```

```
frozenset({1, 2})
```

```
a({"x": 1})
```

```
True
```

```
a({})
```

```
False
```

***SubSetPredicate* attribute** The `SubSetPredicate` attribute can affirm if a value is a subset of a defined set.

```
from galactic_attribute_categorized import SubSetPredicate
a = SubSetPredicate(Key(name="x"), values=frozenset({1, 2}))
print(a)
```

```
x <= {1, 2}
```

```
a({"x": {1}})
```

```
True
```

```
a({"x": {}})
```

```
True
```

```
a({"x": {1, 2, 3}})
```

```
False
```

***SuperSetPredicate* predicate** The `SuperSetPredicate` attribute can affirm if a value is a superset of a defined set.

```
from galactic_attribute_categorized import SuperSetPredicate
a = SuperSetPredicate(Key(name="x"), values=frozenset({1, 2}))
print(a)
```

```
x >= {1, 2}
```

```python
a({"x": {1}})
```

```
False
```

```python
a({"x": {}})
```

```
False
```

```python
a({"x": {1, 2, 3}})
```

```
True
```

3.1.3.2 Descriptions

CategoryDescription A `CategoryDescription` can describe a collection of individuals by calculating the convex hull of categorized predicates.

```python
from galactic_attribute_categorized import CategoryDescription
description = CategoryDescription(Key(name="x"))
attributes = description([{"x": 1}, {"x": 2}])
[str(attribute) for attribute in attributes]
```

```
['x in {1, 2}']
```

3.2 Strategies

3.2.1 Logical Basic Strategy

3.2.1.1 Concepts **F**ormal **C**oncept **A**nalysis consists in finding concepts from a collection of individuals described by boolean attributes. A concept is described by a couple (A, B) where:

- A is a subset of the individual collection;
- B is a a subset of the attributes.

```python
from galactic.concepts import Population
data = {
    0: ["composite", "even", "square"],
    1: ["odd", "square"],
```

```
    2: ["even", "prime"],
    3: ["odd", "prime"],
    4: ["composite", "even", "square"],
    5: ["odd", "prime"],
    6: ["composite", "even"],
    7: ["odd", "prime"],
    8: ["composite", "even"],
    9: ["composite", "odd", "square"],
}
population = Population(data)
list(population)
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
"0" in population
```

```
True
```

```
population.key(data[0])
```

```
'0'
```

Concepts are described by a list of descriptions.

```
from galactic.concepts import Concept
from galactic_attribute_logical import BooleanDescription
from galactic.attributes import Member
descriptions = [
    BooleanDescription(Member(name="composite")),
    BooleanDescription(Member(name="even")),
    BooleanDescription(Member(name="odd")),
    BooleanDescription(Member(name="prime")),
    BooleanDescription(Member(name="square")),
]
concept1 = Concept(
    population=population,
    descriptions=descriptions,
    predicates=[Member(name="composite")]
)
```

```
concept1
```

```
<galactic.concepts.Concept at 0x7f83102ecdc8>
```

```
list(concept1.individuals)
```

```
['0', '4', '6', '8', '9']
```

```
list(concept1.individuals.values())
```

```
[['composite', 'even', 'square'],
 ['composite', 'even', 'square'],
 ['composite', 'even'],
 ['composite', 'even'],
 ['composite', 'odd', 'square']]
```

```
str(concept1.descriptors)
```

```
'composite'
```

```
concept2 = Concept(
    population=population,
    descriptions=descriptions,
    keys=["0", "2", "4"]
)
```

```
list(concept2.individuals.keys())
```

```
['0', '2', '4', '6', '8']
```

Concepts are lattice elements so a unique infimum and a unique supremum exists:

```
infimum = concept1 & concept2
list(infimum.individuals.keys())
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
str(infimum.descriptors)
```

```
''
```

```python
supremum = concept1 | concept2
list(supremum.individuals.keys())
```

```
['0', '4', '6', '8']
```

3.2.1.2 Concept lattices The `Lattice` class is able to extract all concepts from a population by using strategies. In this tutorial, we consider only the boolean case:

```python
from galactic.concepts import Explorer
from galactic_strategy_logical_basic import BooleanStrategy
explorer = Explorer(
    population=population,
    strategies=[
        BooleanStrategy(Member(name="composite")),
        BooleanStrategy(Member(name="even")),
        BooleanStrategy(Member(name="odd")),
        BooleanStrategy(Member(name="prime")),
        BooleanStrategy(Member(name="square")),
    ]
)
```

```python
list(str(concept) for concept in explorer)
```

```
['',
 'composite',
 'even',
 'odd',
 'prime',
 'square',
 'composite and even',
 'composite and square',
 'odd and prime',
 'odd and square',
 'composite and even and square',
 'even and prime',
```

```
'composite and odd and square',
'False and False and False and False']
```

It’s possible to have access to successors and predecessors of a concept:

```python
import yaml
print(
    yaml.dump(
        [
            {
                "concept": list(concept.individuals.keys()),
                "predecessors": [
                    str(predecessor.descriptors)
                    for predecessor in predecessors
                ],
                "successors": [
                    str(successor.descriptors)
                    for successor in successors
                ]
            }
            for concept, (predecessors, successors) in explorer.items()
        ]
    )
)
```


```
- concept:
  - '0'
  - '1'
  - '2'
  - '3'
  - '4'
  - '5'
  - '6'
  - '7'
  - '8'
  - '9'
  predecessors: []
  successors:
  - composite
```

```
    - even
    - odd
    - prime
    - square
- concept:
  - '0'
  - '4'
  - '6'
  - '8'
  - '9'
  predecessors:
  - ''
  successors:
  - composite and even
  - composite and square
- concept:
  - '0'
  - '2'
  - '4'
  - '6'
  - '8'
  predecessors:
  - ''
  successors:
  - composite and even
  - even and prime
- concept:
  - '1'
  - '3'
  - '5'
  - '7'
  - '9'
  predecessors:
  - ''
  successors:
  - odd and prime
  - odd and square
- concept:
```

```
    - '2'
    - '3'
    - '5'
    - '7'
    predecessors:
    - ''
    successors:
    - even and prime
    - odd and prime
  - concept:
    - '0'
    - '1'
    - '4'
    - '9'
    predecessors:
    - ''
    successors:
    - composite and square
    - odd and square
  - concept:
    - '0'
    - '4'
    - '6'
    - '8'
    predecessors:
    - even
    - composite
    successors:
    - composite and even and square
  - concept:
    - '0'
    - '4'
    - '9'
    predecessors:
    - square
    - composite
    successors:
    - composite and even and square
```

```
  - composite and odd and square
- concept:
  - '3'
  - '5'
  - '7'
  predecessors:
  - odd
  - prime
  successors:
  - False and False and False and False and False
- concept:
  - '1'
  - '9'
  predecessors:
  - odd
  - square
  successors:
  - composite and odd and square
- concept:
  - '0'
  - '4'
  predecessors:
  - composite and even
  - composite and square
  successors:
  - False and False and False and False and False
- concept:
  - '2'
  predecessors:
  - even
  - prime
  successors:
  - False and False and False and False and False
- concept:
  - '9'
  predecessors:
  - composite and square
  - odd and square
```

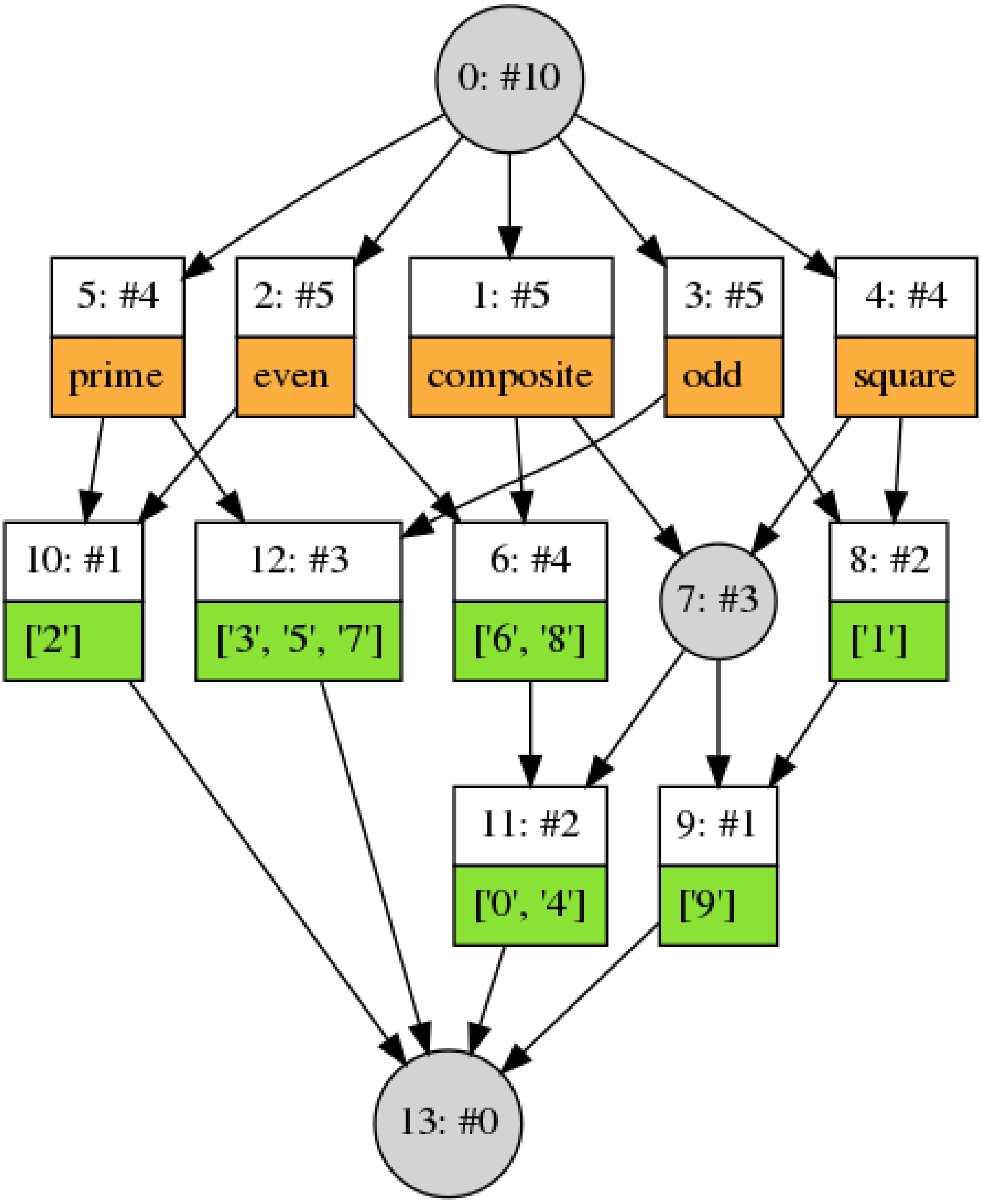
```
  successors:
  - False and False and False and False and False
- concept: []
  predecessors:
  - composite and odd and square
  - composite and even and square
  - odd and prime
  - even and prime
  successors: []
```

The `Lattice` class of the `galactic.concepts` package is able to contruct a lattice:

```python
from galactic.concepts import Lattice
lattice = Lattice(
    population=population,
    strategies=[
        BooleanStrategy(Member(name="composite")),
        BooleanStrategy(Member(name="even")),
        BooleanStrategy(Member(name="odd")),
        BooleanStrategy(Member(name="prime")),
        BooleanStrategy(Member(name="square")),
    ]
)
list(str(concept) for concept in lattice)
```

```
['',
 'composite',
 'even',
 'odd',
 'square',
 'prime',
 'composite and even',
 'composite and square',
 'odd and square',
 'composite and odd and square',
 'even and prime',
 'composite and even and square',
 'odd and prime',
 'False and False and False and False and False']
```

```
lattice
```




```
list(str(concept) for concept in lattice.join_irreducible())
```

```
['composite', 'even', 'odd', 'prime', 'square']
```

```python
list(str(concept) for concept in lattice.meet_irreducible())
```

```
['composite and even',
 'odd and prime',
 'odd and square',
 'composite and even and square',
 'even and prime',
 'composite and odd and square']
```

It's possible to contruct a minimal binary table representing a concept lattice:

```python
{
    str(list(irreducible1.individuals.keys())): [
        str(irreducible2) for irreducible2 in lattice.meet_irreducible()
        if irreducible1 <= irreducible2
    ]
    for irreducible1 in lattice.join_irreducible()
}
```

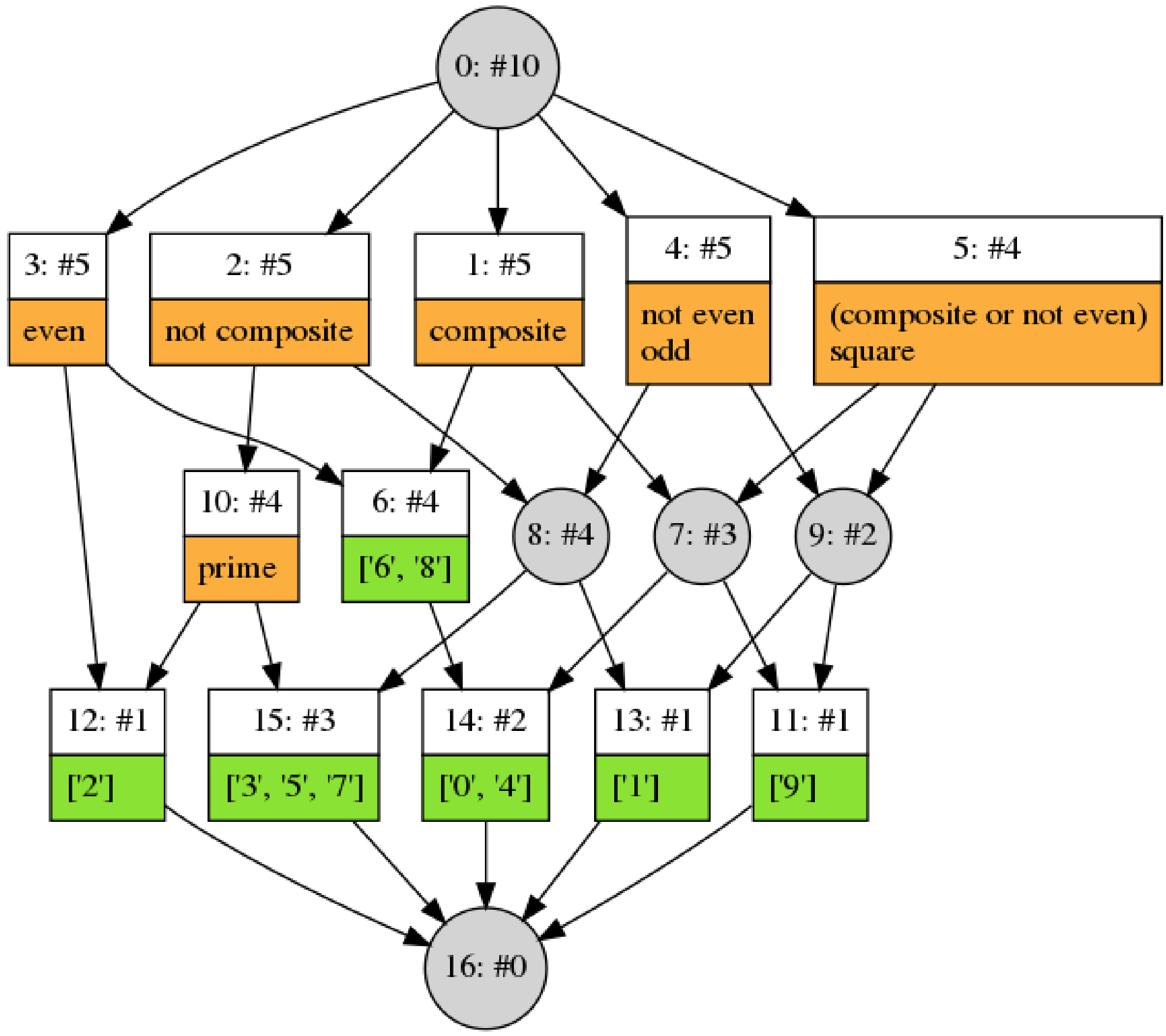
```
{"['0', '4', '6', '8', '9']": ['composite and even',
  'composite and even and square',
  'composite and odd and square'],
 "['0', '2', '4', '6', '8']": ['composite and even',
  'composite and even and square',
  'even and prime'],
 "['1', '3', '5', '7', '9']": ['odd and prime',
  'odd and square',
  'composite and odd and square'],
 "['2', '3', '5', '7']": ['odd and prime', 'even and prime'],
 "['0', '1', '4', '9']": ['odd and square',
  'composite and even and square',
  'composite and odd and square']}
```

There exists a `DualStrategy` able to combine several boolean predicates and their negations.

```python
from galactic_strategy_logical_basic import DualStrategy
```

```
lattice = Lattice(
    population=population,
    strategies=[
        DualStrategy(Member(name="composite"),Member(name="even")),
        BooleanStrategy(Member(name="odd")),
        BooleanStrategy(Member(name="prime")),
        BooleanStrategy(Member(name="square")),
    ]
)
```

```
lattice
```



3.2.2 Numerical Basic Strategy

3.2.2.1 Concepts

```python
from galactic.concepts import Population
data = {
    "Darth Vador": {
        "age": 46,
        "height": 202
    },
    "Boba Fett": {
        "age": 36,
        "height": 183
    },
    "Chewbacca": {
        "age": 204,
        "height": 230
    },
    "Han Solo": {
        "age": 36,
        "height": 180
    },
    "Leia Organa": {
        "age": 23,
        "height": 150
    },
    "Luke Skywalker": {
        "age": 23,
        "height": 172
    },
}
population = Population(data)
list(population)
```

```
['Darth Vador',
 'Boba Fett',
 'Chewbacca',
 'Han Solo',
```

```
 'Leia Organa',
 'Luke Skywalker']
```

```python
"Boba Fett" in population
```

```
True
```

```python
population.key(data["Boba Fett"])
```

```
'Boba Fett'
```

Concepts are described by a list of descriptions.

```python
from galactic.concepts import Concept
from galactic_attribute_numerical import NumericalDescription, lower_limit
from galactic.attributes import Key
descriptions = [
    NumericalDescription(Key(name="age")),
    NumericalDescription(Key(name="height")),
]
concept1 = Concept(
    population=population,
    descriptions=descriptions,
    predicates=[lower_limit(attribute=Key(name="age"),limit=30)]
)
concept1
```

```
<galactic.concepts.Concept at 0x7f8821f31288>
```

```python
list(concept1.individuals)
```

```
['Darth Vador', 'Boba Fett', 'Chewbacca', 'Han Solo']
```

```python
list(concept1.individuals.values())
```

```
[{'age': 46, 'height': 202},
 {'age': 36, 'height': 183},
 {'age': 204, 'height': 230},
 {'age': 36, 'height': 180}]
```

```python
str(concept1.descriptors)
```

```
'age>=36.0 and age<=204 and height>=180.0 and height<=230'
```

```python
concept2 = Concept(
    population=population,
    descriptions=descriptions,
    keys=["Han Solo", "Boba Fett", "Luke Skywalker"]
)
```

```python
list(concept2.individuals.keys())
```

```
['Boba Fett', 'Han Solo', 'Luke Skywalker']
```

```python
str(concept2.descriptors)
```

```
'age>=23.0 and age<=36 and height>=172.0 and height<=183'
```

Concepts are lattice elements so a unique infimum and a unique supremum exists:

```python
infimum = concept1 & concept2
list(infimum.individuals.keys())
```

```
['Darth Vador', 'Boba Fett', 'Chewbacca', 'Han Solo', 'Luke Skywalker']
```

```python
str(infimum.descriptors)
```

```
'age>=23.0 and age<=204 and height>=172.0 and height<=230'
```

```python
supremum = concept1 | concept2
list(supremum.individuals.keys())
```

```
['Boba Fett', 'Han Solo']
```

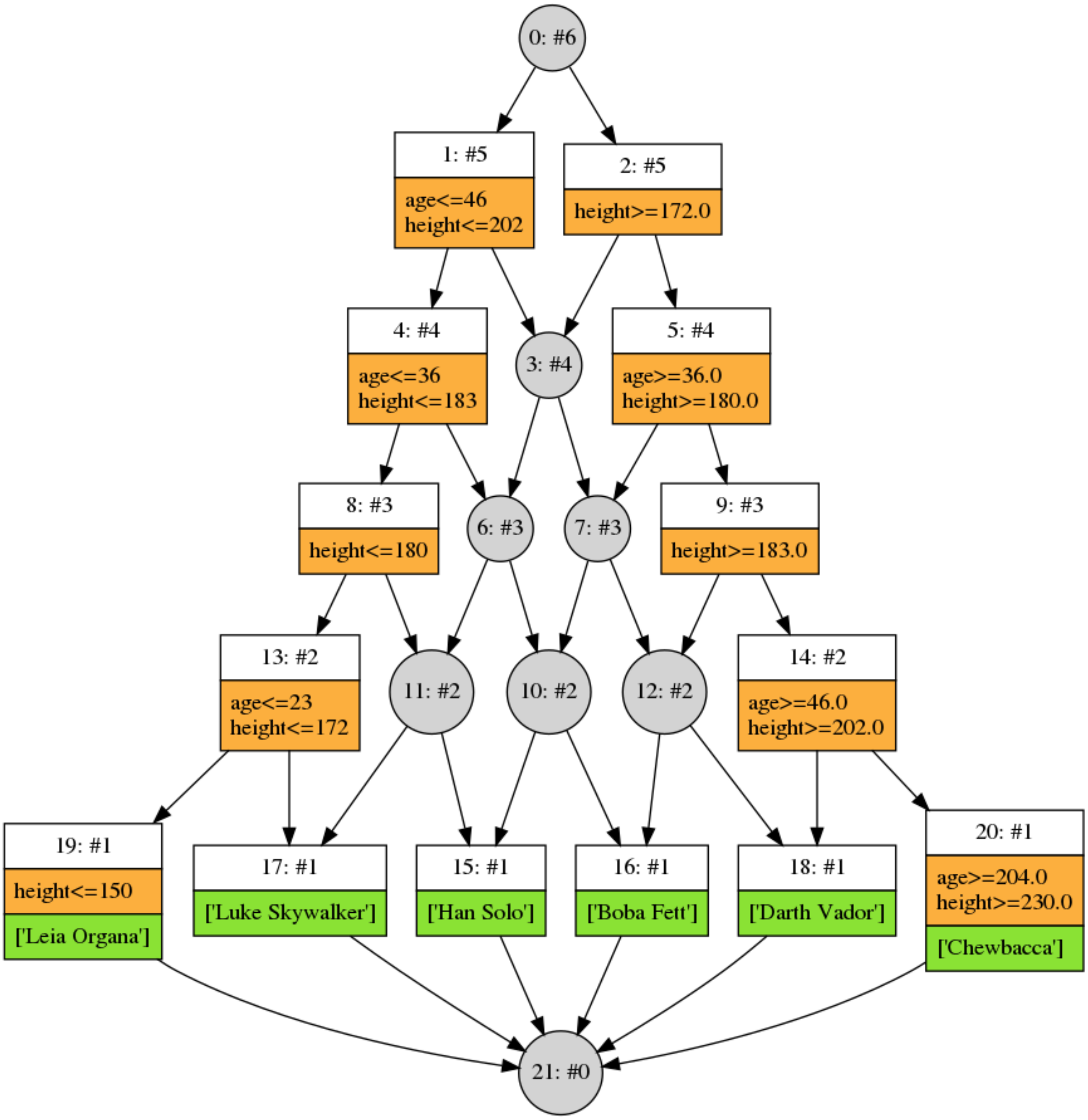
3.2.2.2 Concept lattices The `Lattice` class is able to extract all concepts from a population by using strategies.

```python
from galactic.concepts import Lattice
from galactic_strategy_numerical_basic import NormalStrategy
lattice = Lattice(
    population=population,
    strategies=[
        NormalStrategy(Key(name="age")),
        NormalStrategy(Key(name="height")),
    ]
)
```

```python
list(str(concept) for concept in lattice)
```

```
['age>=23.0 and age<=204 and height>=150.0 and height<=230',
 'age>=23.0 and age<=46 and height>=150.0 and height<=202',
 'age>=23.0 and age<=204 and height>=172.0 and height<=230',
 'age>=23.0 and age<=46 and height>=172.0 and height<=202',
 'age>=23.0 and age<=36 and height>=150.0 and height<=183',
 'age>=36.0 and age<=204 and height>=180.0 and height<=230',
 'age>=23.0 and age<=36 and height>=172.0 and height<=183',
 'age>=36.0 and age<=46 and height>=180.0 and height<=202',
 'age>=23.0 and age<=36 and height>=150.0 and height<=180',
 'age>=36.0 and age<=204 and height>=183.0 and height<=230',
 'age>=36.0 and age<=36 and height>=180.0 and height<=183',
 'age>=23.0 and age<=36 and height>=172.0 and height<=180',
 'age>=36.0 and age<=46 and height>=183.0 and height<=202',
 'age>=23.0 and age<=23 and height>=150.0 and height<=172',
 'age>=46.0 and age<=204 and height>=202.0 and height<=230',
 'age>=36.0 and age<=36 and height>=180.0 and height<=180',
 'age>=36.0 and age<=36 and height>=183.0 and height<=183',
 'age>=23.0 and age<=23 and height>=172.0 and height<=172',
 'age>=46.0 and age<=46 and height>=202.0 and height<=202',
 'age>=23.0 and age<=23 and height>=150.0 and height<=150',
 'age>=204.0 and age<=204 and height>=230.0 and height<=230',
 'False and False']
```

```python
lattice
```




```python
list(str(concept) for concept in lattice.join_irreducible())
```

```
['age>=23.0 and age<=46 and height>=150.0 and height<=202',
 'age>=23.0 and age<=204 and height>=172.0 and height<=230',
 'age>=23.0 and age<=36 and height>=150.0 and height<=183',
 'age>=36.0 and age<=204 and height>=180.0 and height<=230',
 'age>=23.0 and age<=36 and height>=150.0 and height<=180',
 'age>=36.0 and age<=204 and height>=183.0 and height<=230',
 'age>=23.0 and age<=23 and height>=150.0 and height<=172',
```

```
 'age>=46.0 and age<=204 and height>=202.0 and height<=230',
 'age>=23.0 and age<=23 and height>=150.0 and height<=150',
 'age>=204.0 and age<=204 and height>=230.0 and height<=230']
```

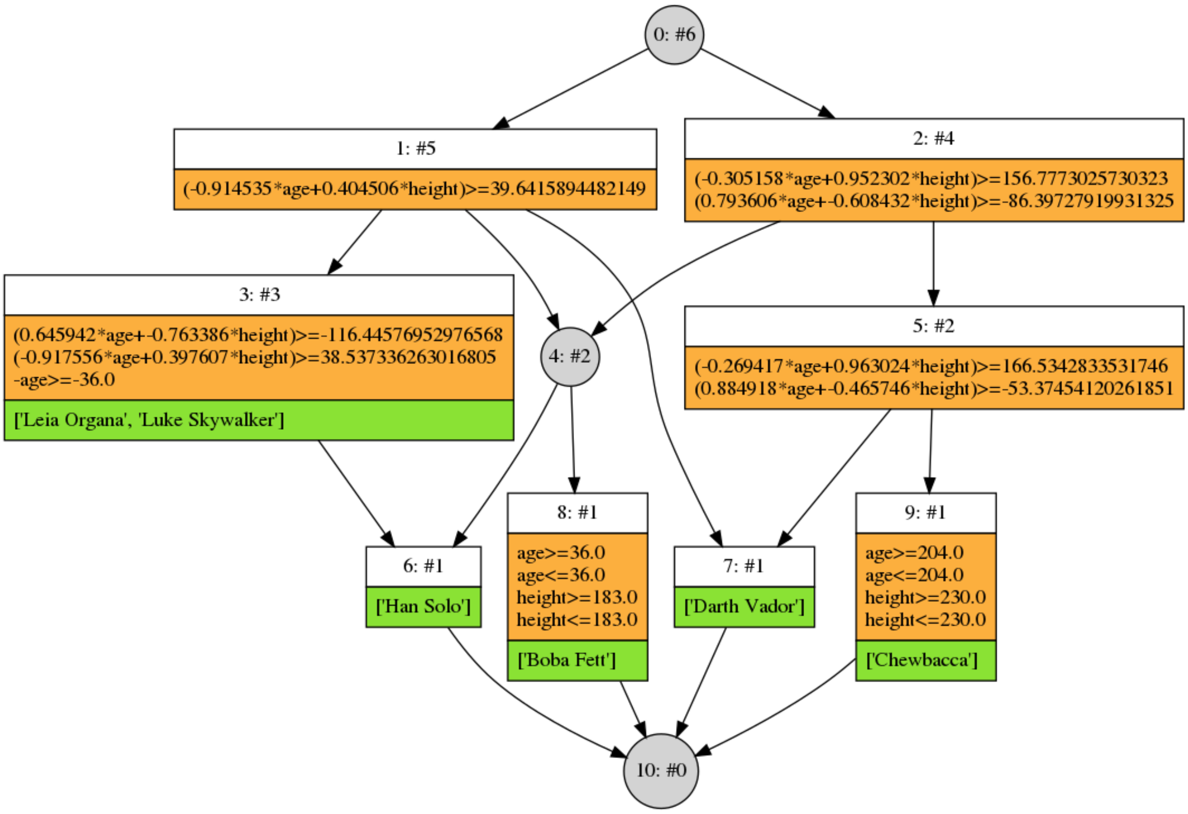
```python
list(str(concept) for concept in lattice.meet_irreducible())
```

```
['age>=36.0 and age<=36 and height>=183.0 and height<=183',
 'age>=36.0 and age<=36 and height>=180.0 and height<=180',
 'age>=23.0 and age<=23 and height>=172.0 and height<=172',
 'age>=23.0 and age<=23 and height>=150.0 and height<=150',
 'age>=46.0 and age<=46 and height>=202.0 and height<=202',
 'age>=204.0 and age<=204 and height>=230.0 and height<=230']
```

The `NormalStrategy` can combine several numerical attributes:

```python
lattice = Lattice(
    population=population,
    strategies=[
        NormalStrategy(Key(name="age"), Key(name="height")),
    ]
)
```

```python
lattice
```



3.2.3 Numerical Quantile Strategy

3.2.3.1 Concept lattices The `Lattice` class is able to extract all concepts from a population by using strategies.

```python
from galactic.concepts import Population
data = {
    "Darth Vador": {
        "age": 46,
        "height": 202
    },
    "Boba Fett": {
        "age": 36,
        "height": 183
    },
    "Chewbacca": {
        "age": 204,
        "height": 230
```

```
    },
    "Han Solo": {
        "age": 36,
        "height": 180
    },
    "Leia Organa": {
        "age": 23,
        "height": 150
    },
    "Luke Skywalker": {
        "age": 23,
        "height": 172
    },
}
population = Population(data)
list(population)
```

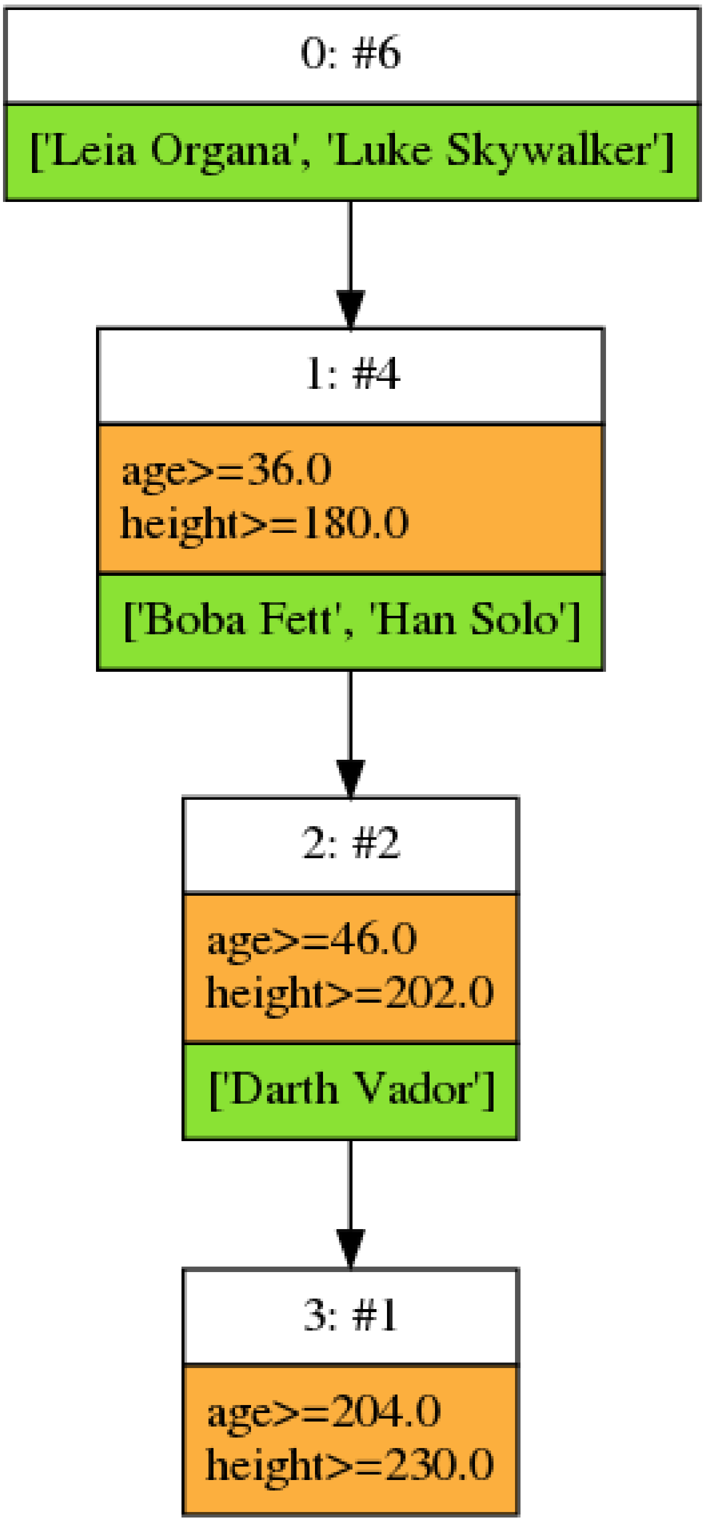
```
['Darth Vador',
 'Boba Fett',
 'Chewbacca',
 'Han Solo',
 'Leia Organa',
 'Luke Skywalker']
```

```
from galactic.attributes import Key
from galactic.concepts import Lattice
from galactic_strategy_numerical_quantile import QuantileStrategy
lattice = Lattice(
    population=population,
    strategies=[
        QuantileStrategy(Key(name="age"),quantile=3),
        QuantileStrategy(Key(name="height"),quantile=3),
    ]
)
```

```
list(str(concept) for concept in lattice)
```

```
['age>=23.0 and age<=204 and height>=150.0 and height<=230',
 'age>=36.0 and age<=204 and height>=180.0 and height<=230',
 'age>=46.0 and age<=204 and height>=202.0 and height<=230',
 'age>=204.0 and age<=204 and height>=230.0 and height<=230']
```

```python
lattice
```




```python
list(str(concept) for concept in lattice.join_irreducible())
```

```
['age>=36.0 and age<=204 and height>=180.0 and height<=230',
 'age>=46.0 and age<=204 and height>=202.0 and height<=230',
 'age>=204.0 and age<=204 and height>=230.0 and height<=230']
```

```python
list(str(concept) for concept in lattice.meet_irreducible())
```

```
['age>=23.0 and age<=204 and height>=150.0 and height<=230',
 'age>=36.0 and age<=204 and height>=180.0 and height<=230',
 'age>=46.0 and age<=204 and height>=202.0 and height<=230']
```

3.2.4 Categorized Strategy

3.2.4.1 Concepts

```python
from galactic.concepts import Population
data = {
    "Jupiter": {"size": "giant", "type": "gaz"},
    "Saturn": {"size": "giant", "type": "gaz"},
    "Uranus": {"size": "giant", "type": "ice"},
    "Neptune": {"size": "giant", "type": "ice"},
    "Venus": {"size": "planet", "type": "telluric"},
    "Earth": {"size": "planet", "type": "telluric"},
    "Mars": {"size": "planet", "type": "telluric"}
}
population = Population(data)
list(population)
```

```
['Jupiter', 'Saturn', 'Uranus', 'Neptune', 'Venus', 'Earth', 'Mars']
```

Concepts are described by a list of descriptions.

```python
from galactic.concepts import Concept
from galactic_attribute_categorized import CategoryDescription
from galactic_strategy_categorized_basic import CategoryPredicate
from galactic.attributes import Key
descriptions = [
    CategoryDescription(Key(name="size")),
    CategoryDescription(Key(name="type")),
```

```python
]
concept1 = Concept(
    population=population,
    descriptions=descriptions,
    predicates=[
        CategoryPredicate(
            attribute=Key(name="size"),
            values=frozenset(["giant"])
        )
    ]
)
concept1
```

```
<galactic.concepts.Concept at 0x7f866c08f5e8>
```

```python
list(concept1.individuals)
```

```
['Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

```python
list(concept1.individuals.values())
```

```
[{'size': 'giant', 'type': 'gaz'},
 {'size': 'giant', 'type': 'gaz'},
 {'size': 'giant', 'type': 'ice'},
 {'size': 'giant', 'type': 'ice'}]
```

```python
str(concept1.descriptors)
```

```
"size in {'giant'} and type in {'ice', 'gaz'}"
```

```python
concept2 = Concept(
    population=population,
    descriptions=descriptions,
    keys=["Uranus", "Neptune", "Earth"]
)
```

```python
list(concept2.individuals.keys())
```

```
['Uranus', 'Neptune', 'Venus', 'Earth', 'Mars']
```

```
str(concept2.descriptors)
```

```
"size in {'planet', 'giant'} and type in {'ice', 'telluric'}"
```

Concepts are lattice elements so a unique infimum and a unique supremum exists:

```
infimum = concept1 & concept2
list(infimum.individuals.keys())
```

```
['Jupiter', 'Saturn', 'Uranus', 'Neptune', 'Venus', 'Earth', 'Mars']
```

```
str(infimum.descriptors)
```

```
"size in {'planet', 'giant'} and type in {'telluric', 'ice', 'gaz'}"
```

```
supremum = concept1 | concept2
list(supremum.individuals.keys())
```

```
['Uranus', 'Neptune']
```

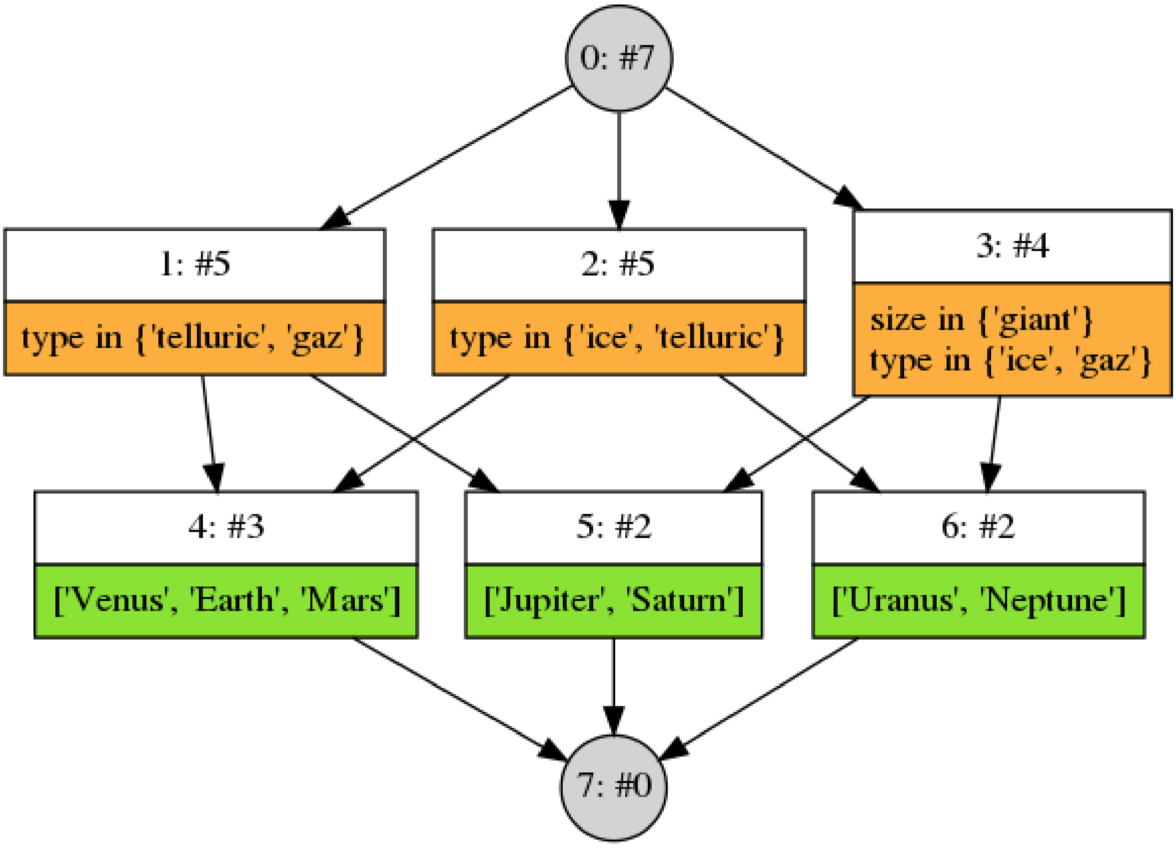
3.2.4.2 Concept lattices The `Lattice` class is able to extract all concepts from a population by using strategies.

```
from galactic.concepts import Lattice
from galactic_strategy_categorized_basic import CategoryStrategy
lattice = Lattice(
    population=population,
    strategies=[
        CategoryStrategy(Key(name="size")),
        CategoryStrategy(Key(name="type")),
    ]
)
```

```
list(str(concept) for concept in lattice)
```

```
["size in {'planet', 'giant'} and type in {'telluric', 'ice', 'gaz'}",
 "size in {'planet', 'giant'} and type in {'telluric', 'gaz'}",
 "size in {'planet', 'giant'} and type in {'ice', 'telluric'}",
 "size in {'giant'} and type in {'ice', 'gaz'}",
 "size in {'planet'} and type in {'telluric'}",
 "size in {'giant'} and type in {'gaz'}",
 "size in {'giant'} and type in {'ice'}",
 'False and False']
```

```python
lattice
```




```python
list(str(concept) for concept in lattice.join_irreducible())
```

```
["size in {'planet', 'giant'} and type in {'telluric', 'gaz'}",
 "size in {'planet', 'giant'} and type in {'ice', 'telluric'}",
 "size in {'giant'} and type in {'ice', 'gaz'}"]
```

```python
list(str(concept) for concept in lattice.meet_irreducible())
```

```
["size in {'planet'} and type in {'telluric'}",
 "size in {'giant'} and type in {'gaz'}",
 "size in {'giant'} and type in {'ice'}"]
```

3.3 Measures

Measure plugins are used in the core strategies `LimitFilter` and `SelectionFilter` which inherit from `Filter`.

3.3.1 *Entropy* measure

A population can be created using a collection of python objects:

```python
from galactic.concepts import Population
individuals = {
    48: {"value": 48, "square": False},
    36: {"value": 36, "square": True},
    64: {"value": 64, "square": True},
    56: {"value": 56, "square": False},
    84: {"value": 84, "square": False}
}
population=Population(individuals)
population
```

```
<galactic.concepts.Population at 0x7f42bd00a9e8>
```

```python
list(population)
```

```
['48', '36', '64', '56', '84']
```

A lattice can be created from a population using a selection filter and a measure based upon the entropy:

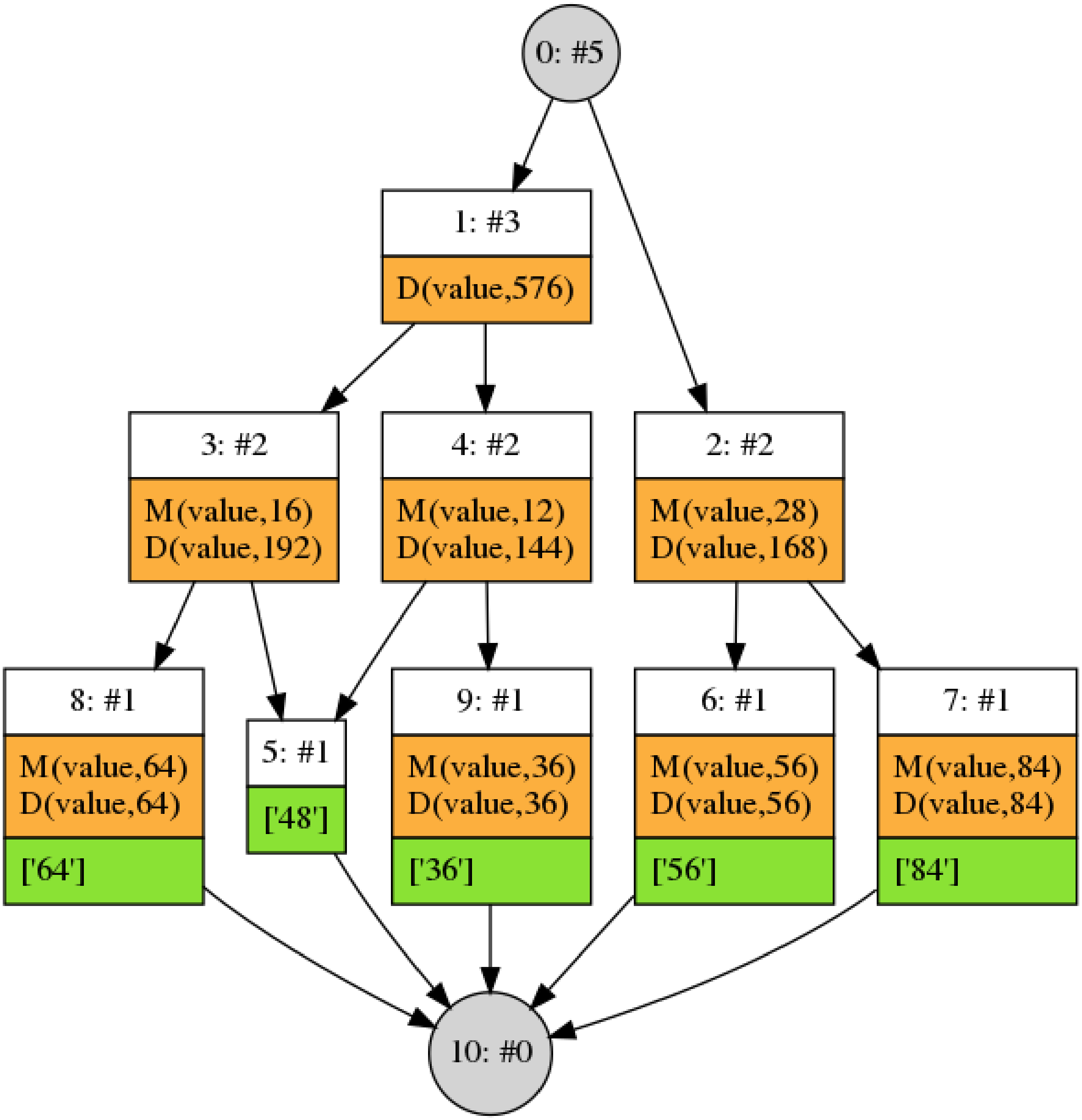
```python
from galactic.concepts import Lattice
from galactic.attributes import Key
```

```python
from galactic.examples.arithmetic.strategies import IntegerStrategy
from galactic.strategies import SelectionFilter
from galactic_measure_entropy import Entropy
strategies = [
    SelectionFilter(
        IntegerStrategy(Key(name="value")),
        measure=Entropy(attribute=Key(name="square")),
        maximize=False
    )
]
lattice = Lattice(population=population, strategies=strategies)
```

```python
list(str(concept) for concept in lattice)
```

```
['M(value,4) and D(value,4032)',
 'M(value,4) and D(value,576)',
 'M(value,28) and D(value,168)',
 'M(value,16) and D(value,192)',
 'M(value,12) and D(value,144)',
 'M(value,48) and D(value,48)',
 'M(value,56) and D(value,56)',
 'M(value,84) and D(value,84)',
 'M(value,64) and D(value,64)',
 'M(value,36) and D(value,36)',
 'False']
```

```python
lattice
```



3.4 Data readers

Data readers are plugins that read population from files which have a specific extension.

3.4.1 Burmeister data reader

Burmeister data reader reads files whose extension is `.cxt`.

```python
from galactic.concepts import Population
import tempfile
from pprint import pprint

input = """\
B

2
2


1
2
a
b
.X
XX
"""

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".cxt") as file:
    file.write(input)
    file.seek(0)
    population = Population.from_file(file)
    pprint({key: sorted(list(value)) for key, value in population.items()})
```

```
{'1': ['b'], '2': ['a', 'b']}
```

3.4.2 *CSV* data reader

CSV data reader reads files whose extension is `.csv`.

```python
from galactic.concepts import Population
import tempfile
from pprint import pprint

input = '''\
age,height
46,202
```

```
36,183
204,230
36,180
23,150
23,172
'''

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".csv") as file:
    file.write(input)
    file.seek(0)
    population = Population.from_file(file)
    pprint({key: dict(value) for key, value in population.items()})
```

```
{'0': {'age': '46', 'height': '202'},
 '1': {'age': '36', 'height': '183'},
 '2': {'age': '204', 'height': '230'},
 '3': {'age': '36', 'height': '180'},
 '4': {'age': '23', 'height': '150'},
 '5': {'age': '23', 'height': '172'}}
```

3.4.3 *FIMI* data reader

FIMI data reader reads files whose extension is `.dat`.

```
from galactic.concepts import Population
from pprint import pprint
import tempfile

input = '''\
1 3
2 4 5
1 2
3 4 5
'''

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".dat") as file:
    file.write(input)
    file.seek(0)
```

```
    population = Population.from_file(file)
    pprint({key: sorted(list(value)) for key, value in population.items()})
```

```
{'0': ['1', '3'], '1': ['2', '4', '5'], '2': ['1', '2'], '3': ['3', '4', '5']}
```

3.4.4 *INI* data reader

INI data reader reads files whose extension is `.ini`.

```python
from galactic.concepts import Population
import tempfile
from pprint import pprint

input = '''\
[#1]
name=Galois
firstname=Évariste
[#2]
name=Wille
firstname=Rudolf
'''

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".ini") as file:
    file.write(input)
    file.seek(0)
    population = Population.from_file(file)
    pprint({key: dict(value) for key, value in population.items()})
```

```
{'#1': {'firstname': 'Évariste', 'name': 'Galois'},
 '#2': {'firstname': 'Rudolf', 'name': 'Wille'}}
```

3.4.5 *JSON* data reader

JSON data reader reads files whose extension is `.json`.

```python
from galactic.concepts import Population
import tempfile
from pprint import pprint
```

```
input = '''\
{
  "#1": {
    "name": "Galois",
    "firstname": "Évariste"
  },
  "#2": {
    "name": "Wille",
    "firstname": "Rudolf"
  }
}
'''

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".json") as file:
    file.write(input)
    file.seek(0)
    population = Population.from_file(file)
    pprint({key: dict(value) for key, value in population.items()})
```

```
{'#1': {'firstname': 'Évariste', 'name': 'Galois'},
 '#2': {'firstname': 'Rudolf', 'name': 'Wille'}}
```

3.4.6 *SLF* data reader

SLF data reader reads files whose extension is `.slf`.

```
from galactic.concepts import Population
import tempfile
from pprint import pprint

input = '''\
[Lattice]
2
3
[Objects]
1 2
[Attributes]
a b c
```

```
[Relation]
0 1 0
1 1 0
'''

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".slf") as file:
    file.write(input)
    file.seek(0)
    population = Population.from_file(file)
    pprint({key: list(sorted(value)) for key, value in population.items()})
```

```
{'1': ['b'], '2': ['a', 'b']}
```

3.4.7 *TEXT* data reader

TEXT data reader reads files whose extension is `.txt`.

```
from galactic.concepts import Population
import tempfile
from pprint import pprint

input = '''\
Observations: 1 2 3 4
Attributes: a b c d e
1: a c
2: a b
3: b d e
4: c e
'''

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".txt") as file:
    file.write(input)
    file.seek(0)
    population = Population.from_file(file)
    pprint({key: list(sorted(value)) for key, value in population.items()})
```

```
{'1': ['a', 'c'], '2': ['a', 'b'], '3': ['b', 'd', 'e'], '4': ['c', 'e']}
```

3.4.8 *TOML* data reader

TOML data reader reads files whose extension is `.toml`.

```
from galactic.concepts import Population
import tempfile
from pprint import pprint

input = '''\
# This is a TOML document.
[individual1]
    name="Galois"
    firstname="Évariste"
[individual2]
    name="Wille"
    firstname="Rudolf"
'''

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".toml") as file:
    file.write(input)
    file.seek(0)
    population = Population.from_file(file)
    pprint({key: value for key, value in population.items()})
```

```
{'individual1': {'firstname': 'Évariste', 'name': 'Galois'},
 'individual2': {'firstname': 'Rudolf', 'name': 'Wille'}}
```

3.4.9 *YAML* data reader

YAML data reader reads files whose extension is `.yaml` or `.yml`.

```
from galactic.concepts import Population
import tempfile
from pprint import pprint

input = '''\
# This is a YAML document.
- name: Galois
```

```
  firstname: Évariste
- name: Wille
  firstname: Rudolf
'''

with tempfile.NamedTemporaryFile(mode="w+t", suffix=".yaml") as file:
    file.write(input)
    file.seek(0)
    population = Population.from_file(file)
    pprint({key: value for key, value in population.items()})
```

```
{'0': {'firstname': 'Évariste', 'name': 'Galois'},
 '1': {'firstname': 'Rudolf', 'name': 'Wille'}}
```