Towards Universally Accessible SAT Technology

Alexey Ignatiev, Zi Li Tan, Christos Karamanos

23rd of August, 2024

MONASH University

Motivation

The SAT disrupion

- **SAT is NP-complete**

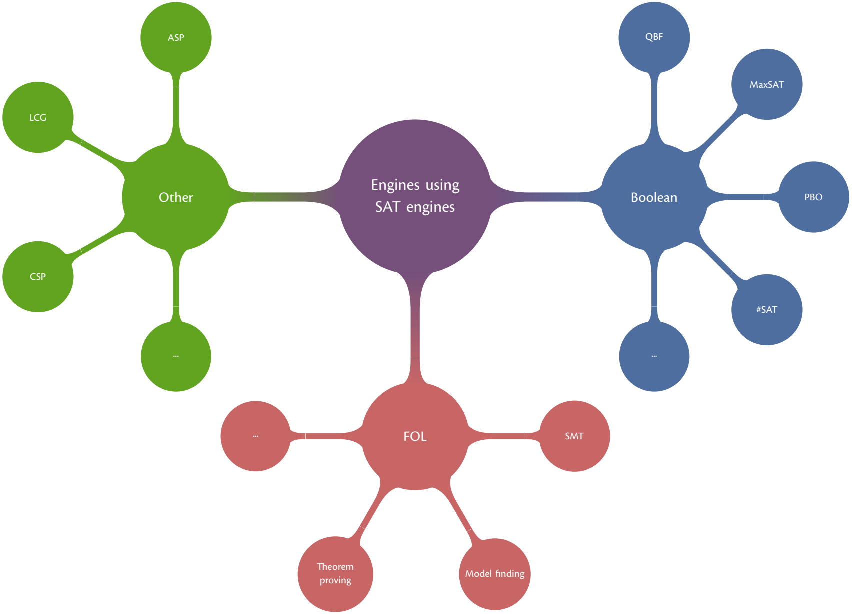
The SAT disrupion

- **SAT is NP-complete**
  - **But CDCL SAT solving is a success story of Computer Science**
  - **Hundreds (thousands?) of practical applications**

SAT is the engines’ engine


Logic Programming
Automated Theorem Proving
Answer Set Programming
CDCL-inspired engine
Reasoning
Interpolation
Instantiation
stable models
Superposition
Resolution
CDCL for PBO
Pseudo-Boolean Optimization
Mixed Integer Programming
Branch & Bound
CDCL
SAT
MaxSAT
Arrays
Bitvectors
Optimization modulo theories
CDCL(T)
cutting planes
LCG
global constraints
custom search
propagators
Satisfiability Modulo Theories
Constraint Satisfaction and Optimization
Integers
EUF
Reals
MiniZinc
Local search

SAT is the engines' engine



How to solve problems with SAT?

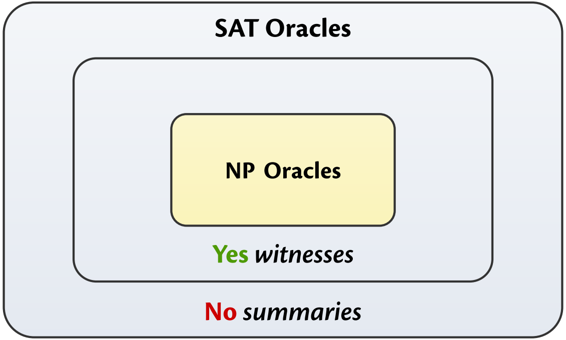
- Use SAT solvers as oracles
- Should be quick to prototype
- Should be reasonably efficient

How to solve problems with SAT?

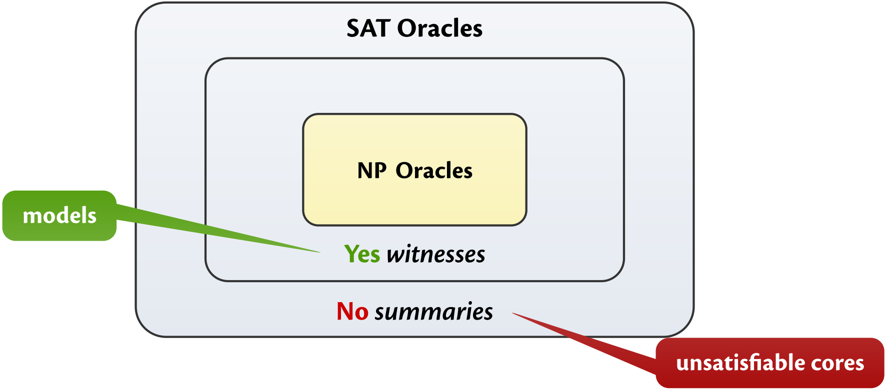
- Use SAT solvers as oracles
- Should be quick to prototype
- Should be reasonably efficient
- Should enable fiddling with the algorithms
- Avoid steep learning curves
- Combine with other technologies
- ...

SAT Oracles

What are SAT oracles?



What are SAT oracles?



Where are we using SAT oracles?

MaxSAT

Where are we using SAT oracles?

MaxSAT

MinSAT; Maximal Falsifiability

Where are we using SAT oracles?

MaxSAT

MCS Extraction
& Enumeration

MinSAT; Maximal
Falsifiability

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Where are we using SAT oracles?

MaxSAT	MCS Extraction & Enumeration	MUS Extraction & Enumeration
MinSAT; Maximal Falsifiability		MSMP

Where are we using SAT oracles?

MaxSAT	MCS Extraction & Enumeration	MUS Extraction & Enumeration
MinSAT; Maximal Falsifiability	Lean Kernels; Backbones	MSMP

Where are we using SAT oracles?

MaxSAT	MCS Extraction & Enumeration	MUS Extraction & Enumeration
MinSAT; Maximal Falsifiability	Lean Kernels; Backbones	MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Where are we using SAT oracles?

MaxSAT	MCS Extraction & Enumeration	MUS Extraction & Enumeration
MinSAT; Maximal Falsifiability	Lean Kernels; Backbones	MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

QBF & QMaxSAT

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

QBF & QMaxSAT

MES Extraction

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

QBF & QMaxSAT

MES Extraction

Maximum Cliques

Where are we using SAT oracles?

MaxSAT	MCS Extraction & Enumeration	MUS Extraction & Enumeration
MinSAT; Maximal Falsifiability	Lean Kernels; Backbones	MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

QBF & QMaxSAT	MES Extraction	Maximum Cliques	Explainable AI

Some challenges

- Low-level (C/C++, even Java) implementations are important:
  - **Iterative** SAT solving
  - Often using **incremental** SAT
  - Need to analyze **models**
  - Need to extract **unsatisfiable cores**
  - **Many practical successes**
- But low-level implementations can be **problematic**:
  - Development time
  - Error prone
  - Difficult to maintain & change
  - ...

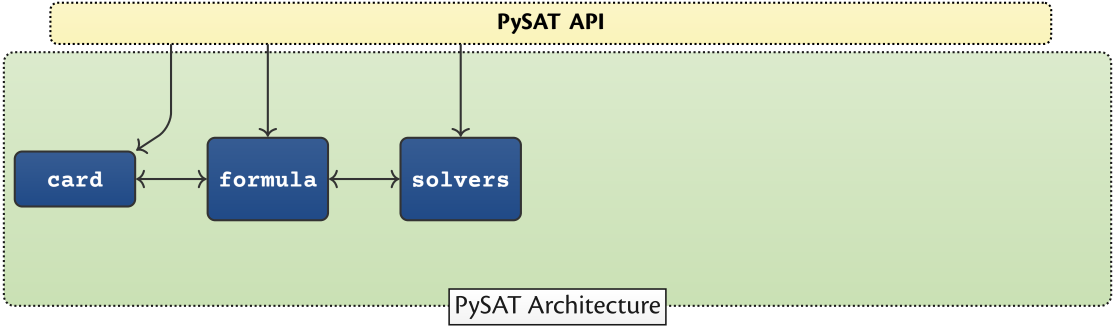
PySAT was introduced in 2018

PySAT was introduced in 2018

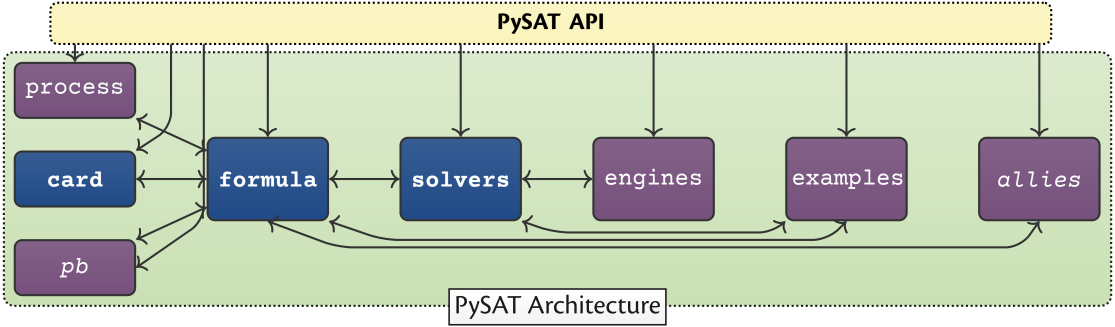
widely used in practice since then!

PySAT

Overview of PySAT

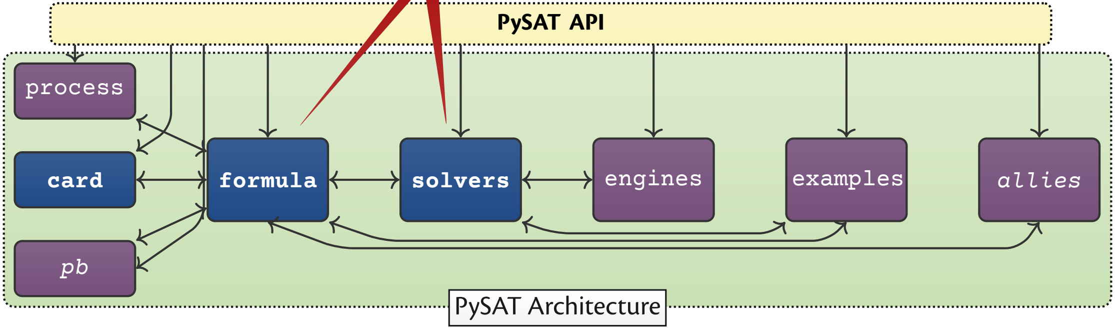


Overview of PySAT

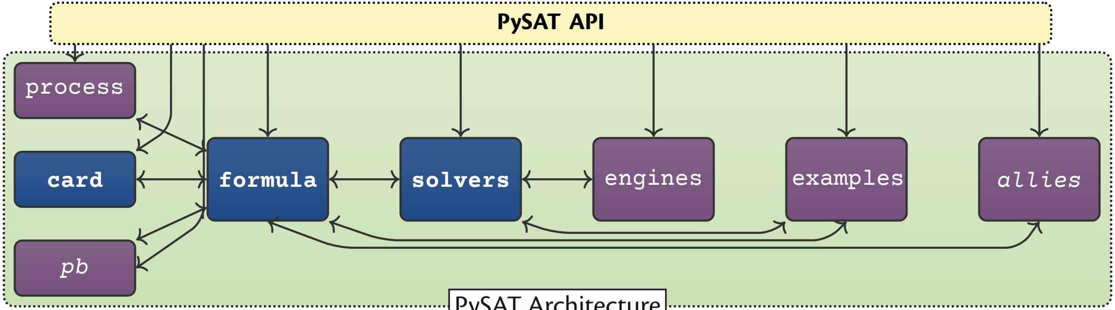


Overview of PySAT

extensively updated since 2018



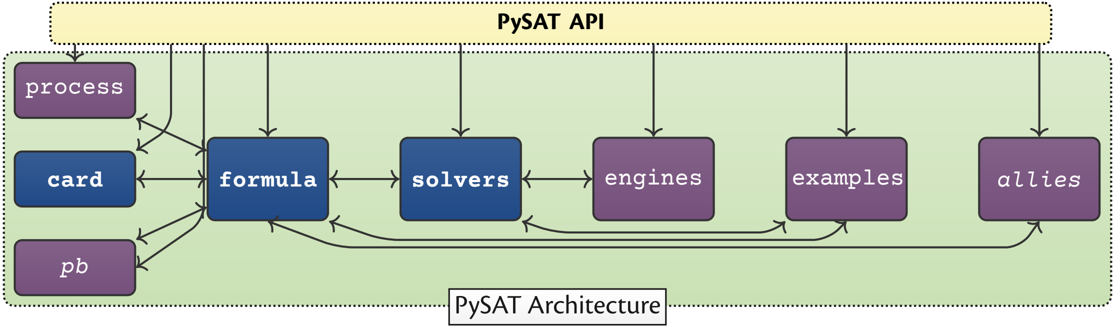
Overview of PySAT



PySAT Architecture

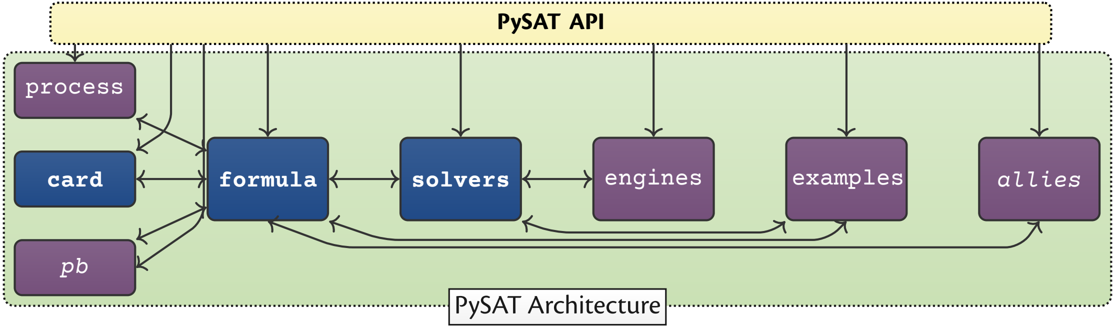
- **Open source, available on github**

Overview of PySAT



- **Open source, available on github**
- Comprehensive list of **SAT solvers**
- Comprehensive list of **cardinality encodings**
- Fairly **comprehensive documentation**
- A number of **examples**

Overview of PySAT



- **Open source, available on github**
- Comprehensive list of **SAT solvers**
- Comprehensive list of **cardinality encodings**
- Fairly **comprehensive documentation**
- A number of **examples**
- External reasoning engines
- Formula (pre-)processing
- *Optional (third-party)* ***pb*** *and* ***allies*** *modules*

Formula manipulation (`pysat.formula` module)

Features
CNF & Weighted CNF (WCNF)
Formula input / output
Append clauses to formula
Negate CNF formulas
Translate between CNF and WCNF
Rudimentary support of non-clausal formulas
ID manager

- `https://pysathq.github.io/docs/html/api/formula`

Clausal formulas in `pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq 1$ while $x_5 \triangleq 5$

Clausal formulas in `pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq 1$ while $x_5 \triangleq 5$
- **a literal is an integer**
  - e.g. $\neg x_1 \triangleq -1$ while $x_5 \triangleq 5$

Clausal formulas in `pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq$ `1` while $x_5 \triangleq$ `5`
- **a literal is an integer**
  - e.g. $\neg x_1 \triangleq$ `-1` while $x_5 \triangleq$ `5`
- **a clause is a *list* of literals**
  - clause $(\neg x_3 \vee x_2) \triangleq$ `[-3, 2]`

Clausal formulas in `pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq$ `1` while $x_5 \triangleq$ `5`
- **a literal is an integer**
  - e.g. $\neg x_1 \triangleq$ `-1` while $x_5 \triangleq$ `5`
- **a clause is a *list* of literals**
  - clause $(\neg x_3 \vee x_2) \triangleq$ `[-3, 2]`
- **a CNF formula is an object of class `CNF`**
  - i.e. basically it is a *list of clauses*

Clausal formulas in `pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq 1$ while $x_5 \triangleq 5$
- **a literal is an integer**
  - e.g. $\neg x_1 \triangleq -1$ while $x_5 \triangleq 5$
- **a clause is a *list* of literals**
  - clause $(\neg x_3 \vee x_2) \triangleq$ `[-3, 2]`
- **a CNF formula is an object of class `CNF`**
  - i.e. basically it is a *list of clauses*
- `IDPool` is a variable ID manager

```
>>> from pysat.formula import IDPool
>>> vpool = IDPool(occupied=[[12, 18], [3, 10]])
>>>
>>> # creating 5 unique variables for the following strings ('v1', 'v2', ..., 'v5')
>>> for i in range(5):
...     print(vpool.id('v{0}'.format(i + 1)))
1
2
11
19
20
```

pysat.formula.CNF basic example

```
>>> from pysat.formula import CNF
>>>
>>> cnf = CNF(from_clauses=[[-1, 2], [3]])
>>> cnf.append([-3, 4])
>>>
>>> print(cnf.clauses)
[[-1, 2], [3], [-3, 4]]
>>>
>>> neg = cnf.negate()
>>> print(neg.clauses)
[[1, -5], [-2, -5], [-1, 2, 5], [3, -6], [-4, -6], [-3, 4, 6], [5, -3, 6]]
>>>
>>> print(neg.auxvars)
[5, -3, 6]
```

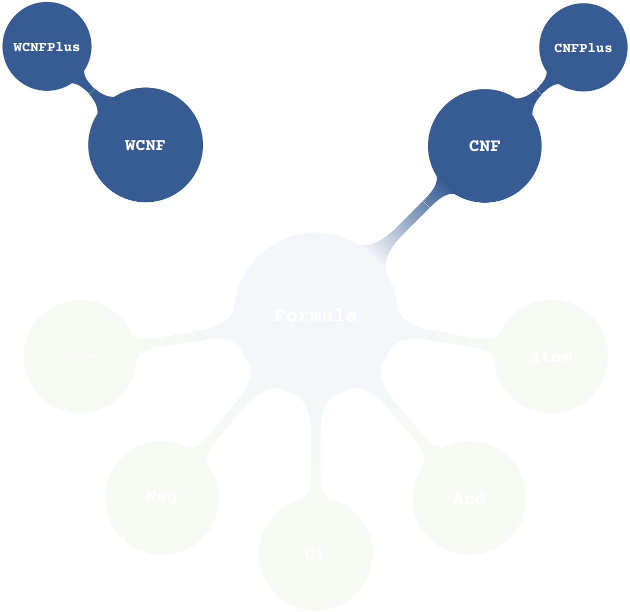
pysat.formula.CNF basic example

```
>>> from pysat.formula import CNF
>>>
>>> cnf = CNF(from_clauses=[[-1, 2], [3]])
>>> cnf.append([-3, 4])
>>>
>>> print(cnf.clauses)
[[-1, 2], [3], [-3, 4]]
>>>
>>> neg = cnf.negate()
>>> print(neg.clauses)
[[1, -5], [-2, -5], [-1, 2, 5], [3, -6], [-4, -6], [-3, 4, 6], [5, -3, 6]]
>>>
>>> print(neg.auxvars)
[5, -3, 6]
```

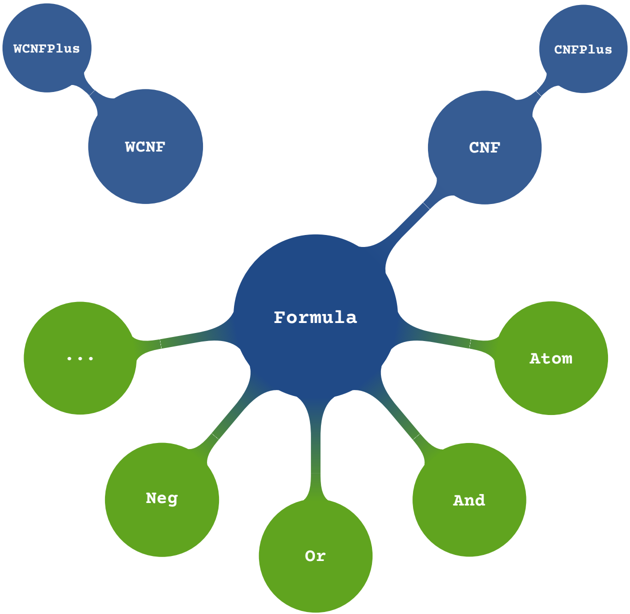
- **much more functionality!**

arbitrary Boolean formulas

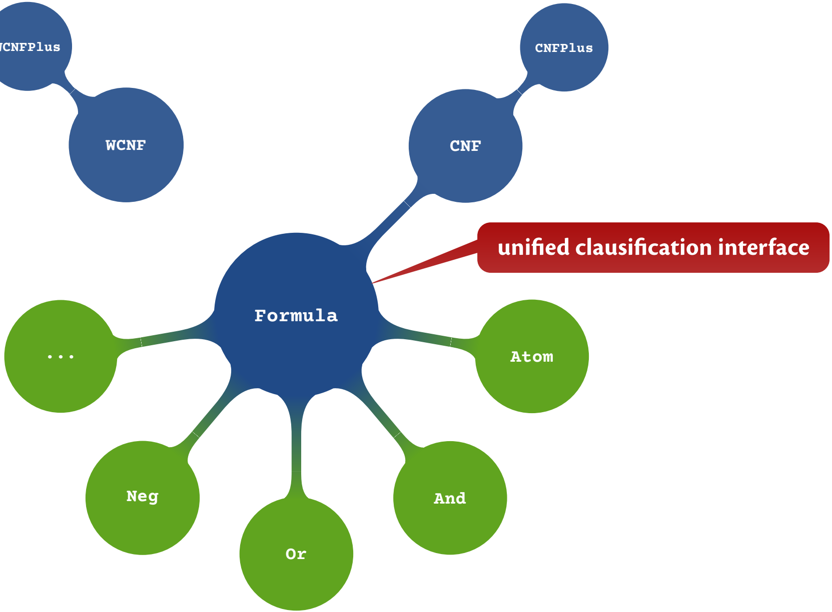
Handling arbitrary Boolean formulas



Handling arbitrary Boolean formulas



Handling arbitrary Boolean formulas



Non-clausal example

```
>>> from pysat.formula import Atom
>>>
>>> x, y, z = [Atom(c) for c in 'xyz']
>>>
>>> form = ~(~x >> y) | (x & z)
>>>
>>> print([cl for cl in form])
>>>
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4], [-3, 5]]
```

Non-clausal example

```
>>> from pysat.formula import Atom
>>>
>>> x, y, z = [Atom(c) for c in 'xyz']
>>>
>>> form = ~(~x >> y) | (x & z)
>>>
>>> print([cl for cl in form])
>>>
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4], [-3, 5]]
```

creating 3 Boolean variables

Non-clausal example

```
>>> from pysat.formula import Atom
>>>
>>> x, y, z = [Atom(c) for c in 'xyz']
>>>
>>> form = ~(~x >> y) | (x & z)
>>>
>>> print([cl for cl in form])
>>>
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4], [-3, 5]]
```

creating 3 Boolean variables

non-clausal formula $\neg(\neg x \rightarrow y) \vee (x \wedge z)$

Non-clausal example

```
>>> from pysat.formula import Atom
>>>
>>> x, y, z = [Atom(c) for c in 'xyz']
>>>
>>> form = ~(~x >> y) | (x & z)
>>>
>>> print([cl for cl in form])
>>>
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4], [-3, 5]]
```

creating 3 Boolean variables

non-clausal formula $\neg(\neg x \rightarrow y) \vee (x \wedge z)$

clausifying *on the fly*

Another non-clausal example

```
>>> from pysat.formula import *
>>> from pysat.solvers import Solver
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> form = (x @ y) ^ z
>>>
>>> with Solver(bootstrap_with=form) as solver:
...     for model in solver.enum_models():
...         # mapping the model back to atoms
...         atoms = Formula.formulas(model, atoms_only=True)
...         print([str(lit) for lit in atoms])
...         # blocking the model
...         solver.add_clause([-lit for lit in Formula.literals(atoms)])
...
['x', '~y', 'z']
['x', 'y', '~z']
['~x', 'y', 'z']
['~x', '~y', '~z']
```

Another non-clausal example

```
>>> from pysat.formula import *
>>> from pysat.solvers import Solver
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> form = (x @ y) ^ z
>>>
>>> with Solver(bootstrap_with=form) as solver:
...     for model in solver.enum_models():
...         # mapping the model back to atoms
...         atoms = Formula.formulas(model, atoms_only=True)
...         print([str(lit) for lit in atoms])
...         # blocking the model
...         solver.add_clause([-lit for lit in Formula.literals(atoms)])
...
['x', '~y', 'z']
['x', 'y', '~z']
['~x', 'y', 'z']
['~x', '~y', '~z']
```

formula $(x \leftrightarrow y) \oplus z$

Another non-clausal example

```
>>> from pysat.formula import *
>>> from pysat.solvers import Solver
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> form = (x @ y) ^ z
>>>
>>> with Solver(bootstrap_with=form) as solver:
...     for model in solver.enum_models():
...         # mapping the model back to atoms
...         atoms = Formula.formulas(model, atoms_only=True)
...         print([str(lit) for lit in atoms])
...         # blocking the model
...         solver.add_clause([-lit for lit in Formula.literals(atoms)])
...
['x', '~y', 'z']
['x', 'y', '~z']
['~x', 'y', 'z']
['~x', '~y', '~z']
```

formula $(x \leftrightarrow y) \oplus z$

solver clausifies it on the fly

Another non-clausal example

```
>>> from pysat.formula import *
>>> from pysat.solvers import Solver
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> form = (x @ y) ^ z
>>>
>>> with Solver(bootstrap_with=form) as solver:
...     for model in solver.enum_models():
...         # mapping the model back to atoms
...         atoms = Formula.formulas(model, atoms_only=True)
...         print([str(lit) for lit in atoms])
...         # blocking the model
...         solver.add_clause([-lit for lit in Formula.literals(atoms)])
...
['x', '~y', 'z']
['x', 'y', '~z']
['~x', 'y', 'z']
['~x', '~y', '~z']
```

formula $(x \leftrightarrow y) \oplus z$

solver clausifies it on the fly

enumerating all models

Yet another non-clausal example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) | z
>>>
>>> print(a.simplified(assumptions=[z]))
T
>>>
>>> print(a.simplified(assumptions=[~z]))
x @ y
>>>
b = a ^ Atom('p')  # a more complex formula
>>>
print(b.simplified(assumptions=[x, ~Atom('p')]))
y | z
```

Yet another non-clausal example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) | z
>>>
>>> print(a.simplified(assumptions=[z]))
T
>>>
>>> print(a.simplified(assumptions=[~z]))
x @ y
>>>
b = a ^ Atom('p')  # a more complex formula
>>>
print(b.simplified(assumptions=[x, ~Atom('p')]))
y | z
```

formula $a \triangleq (x \leftrightarrow y) \vee z$

Yet another non-clausal example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) | z
>>>
>>> print(a.simplified(assumptions=[z]))
T
>>>
>>> print(a.simplified(assumptions=[~z]))
x @ y
>>>
b = a ^ Atom('p')  # a more complex formula
>>>
print(b.simplified(assumptions=[x, ~Atom('p')]))
y | z
```

formula $a \triangleq (x \leftrightarrow y) \vee z$

assigning literal z results in $\top$

Yet another non-clausal example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) | z
>>>
>>> print(a.simplified(assumptions=[z]))
T
>>>
>>> print(a.simplified(assumptions=[~z]))
x @ y
>>>
b = a ^ Atom('p')  # a more complex formula
>>>
print(b.simplified(assumptions=[x, ~Atom('p')]))
y | z
```

formula $a \triangleq (x \leftrightarrow y) \vee z$

assigning literal z results in $\top$

assigning literal $\neg z$ results in $x \leftrightarrow y$

Yet another non-clausal example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) | z
>>>
>>> print(a.simplified(assumptions=[z]))
T
>>>
>>> print(a.simplified(assumptions=[~z]))
x @ y
>>>
b = a ^ Atom('p')  # a more complex formula
>>>
print(b.simplified(assumptions=[x, ~Atom('p')]))
y | z
```

formula $a \triangleq (x \leftrightarrow y) \vee z$

assigning literal z results in $\top$

assigning literal $\neg z$ results in $x \leftrightarrow y$

$b \triangleq a \oplus p$

assigning literals x and $\neg p$ results in $y \vee z$

many more solvers since 2018!

many more solvers since 2018!

+ *extended interface!*

new module `process`

new module `process`

offers formula pre-processing

produces equisatisfiable formulas

Formula processing example

```
>>> from pysat.process import Processor
>>>
>>> proc = Processor(bootstrap_with=[[-1, 2], [-2, 3], [-1, -3], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[[]], False # result contains an empty clause and is unsatisfiable
```

Formula processing example

formula $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1)$

```
>>> from pysat.process import Processor
>>>
>>> proc = Processor(bootstrap_with=[[-1, 2], [-2, 3], [-1, -3], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[[]], False # result contains an empty clause and is unsatisfiable
```

Formula processing example

formula $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1)$

```
>>> from pysat.process import Processor
>>>
>>> proc = Processor(bootstrap_with=[[-1, 2], [-2, 3], [-1, -3], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[[]], False # result contains an empty clause and is unsatisfiable
```

running the processor produces a new formula

Formula processing example

formula $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1)$

```
>>> from pysat.process import Processor
>>>
>>> proc = Processor(bootstrap_with=[[-1, 2], [-2, 3], [-1, -3], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[[]], False # result contains an empty clause and is unsatisfiable
```

running the processor produces a new formula

result formula contains an **empty clause** and has **status** `False`

Formula processing example

formula $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1)$

```
>>> from pysat.process import Processor
>>>
>>> proc = Processor(bootstrap_with=[[-1, 2], [-2, 3], [-1, -3], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[[]], False # result contains an empty clause and is unsatisfiable
```

running the processor produces a new formula

result formula contains an **empty clause** and has **status** `False`

hence, it is unsatisfiable

Another formula processing example

```
>>> from pysat.process import Processor
>>> from pysat.solvers import Solver
>>>
>>> proc = Processor(bootstrap_with=[[-1, -2], [1, 2], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[], True # result has no clauses and is not found to be unsatisfiable
>>>
>>> with Solver(bootstrap_with=processed) as solver:
...     st, mod = solver.solve(), solver.get_model()
...     print('status: {0}, model: {1}'.format(st, proc.restore(mod)))
status: True, model: [1, -2] # result is confirmed to be satisfiable
                             # and the correct model is restored

```

Another formula processing example

formula $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1)$

```
>>> from pysat.process import Processor
>>> from pysat.solvers import Solver
>>>
>>> proc = Processor(bootstrap_with=[[-1, -2], [1, 2], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[], True # result has no clauses and is not found to be unsatisfiable
>>>
>>> with Solver(bootstrap_with=processed) as solver:
...     st, mod = solver.solve(), solver.get_model()
...     print('status: {0}, model: {1}'.format(st, proc.restore(mod)))
status: True, model: [1, -2] # result is confirmed to be satisfiable
                             # and the correct model is restored

```

Another formula processing example

```
>>> from pysat.process import Processor
>>> from pysat.solvers import Solver
>>>
>>> proc = Processor(bootstrap_with=[[-1, -2], [1, 2], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[], True # result has no clauses and is not found to be unsatisfiable
>>>
>>> with Solver(bootstrap_with=processed) as solver:
...     st, mod = solver.solve(), solver.get_model()
...     print('status: {0}, model: {1}'.format(st, proc.restore(mod)))
status: True, model: [1, -2] # result is confirmed to be satisfiable
                              # and the correct model is restored

```

formula $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1)$

processed formula

Another formula processing example

```
>>> from pysat.process import Processor
>>> from pysat.solvers import Solver
>>>
>>> proc = Processor(bootstrap_with=[[-1, -2], [1, 2], [1]])
>>> processed = proc.process()
>>>
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[], True # result has no clauses and is not found to be unsatisfiable
>>>
>>> with Solver(bootstrap_with=processed) as solver:
...     st, mod = solver.solve(), solver.get_model()
...     print('status: {0}, model: {1}'.format(st, proc.restore(mod)))
status: True, model: [1, -2] # result is confirmed to be satisfiable
                             # and the correct model is restored
```

formula $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1)$

processed formula

we can **restore** a model for the original formula

external engines

external engines

through IPASIR-UP interface

Engines interface

```
class Propagator(object):
    def on_assignment(self, lit: int, fixed: bool = False) -> None:
        pass      # receive a new literal assigned by the solver

    def on_new_level(self) -> None:
        pass      # get notified about a new decision level

    def on_backtrack(self, to: int) -> None:
        pass      # process backtracking to a given level

    def check_model(self, model: list[int]) -> bool:
        pass      # check if a given assignment is indeed a model

    def decide(self) -> int:
        return 0  # make a decision and (if any) inform the solver

    def propagate(self) -> list[int]:
        return [] # propagate and return inferred literals (if any)

    def provide_reason(self, lit: int) -> list[int]:
        pass      # explain why a given literal was propagated

    def add_clause(self) -> list[int]:
        return [] # add an(y) external clause to the solver
```

Engines interface

an engine **must inherit** from this abstract class

```python
class Propagator(object):
    def on_assignment(self, lit: int, fixed: bool = False) -> None:
        pass      # receive a new literal assigned by the solver

    def on_new_level(self) -> None:
        pass      # get notified about a new decision level

    def on_backtrack(self, to: int) -> None:
        pass      # process backtracking to a given level

    def check_model(self, model: list[int]) -> bool:
        pass      # check if a given assignment is indeed a model

    def decide(self) -> int:
        return 0  # make a decision and (if any) inform the solver

    def propagate(self) -> list[int]:
        return [] # propagate and return inferred literals (if any)

    def provide_reason(self, lit: int) -> list[int]:
        pass      # explain why a given literal was propagated

    def add_clause(self) -> list[int]:
        return [] # add an(y) external clause to the solver
```

Engines interface

an engine **must inherit** from this abstract class

```python
class Propagator(object):
    def on_assignment(self, lit: int, fixed: bool = False) -> None:
        pass      # receive a new literal assigned by the solver

    def on_new_level(self) -> None:
        pass      # get notified about a new decision level

    def on_backtrack(self, to: int) -> None:
        pass      # process backtracking to a given level

    def check_model(self, model: list[int]) -> bool:
        pass      # check if a given assignment is indeed a model

    def decide(self) -> int:
        return 0  # make a decision and (if any) inform the solver

    def propagate(self) -> list[int]:
        return [] # propagate and return inferred literals (if any)

    def provide_reason(self, lit: int) -> list[int]:
        pass      # explain why a given literal was propagated

    def add_clause(self) -> list[int]:
        return [] # add an(y) external clause to the solver
```

multiple literals at once

in contrast to IPASIR-UP!

Engines interface

an engine **must inherit** from this abstract class

```python
class Propagator(object):
    def on_assignment(self, lit: int, fixed: bool = False) -> None:
        pass      # receive a new literal assigned by the solver

    def on_new_level(self) -> None:
        pass      # get notified about a new decision level

    def on_backtrack(self, to: int) -> None:
        pass      # process backtracking to a given level

    def check_model(self, model: list[int]) -> bool:
        pass      # check if a given assignment is indeed a model

    def decide(self) -> int:
        return 0  # make a decision and (if any) inform the solver

    def propagate(self) -> list[int]:
        return [] # propagate and return inferred literals (if any)

    def provide_reason(self, lit: int) -> list[int]:
        pass      # explain why a given literal was propagated

    def add_clause(self) -> list[int]:
        return [] # add an(y) external clause to the solver
```

to be implemented in Python!

multiple literals at once

in contrast to IPASIR-UP!

A quick example

```python
class MyPowerfulEngine(Propagator):
    # define all the methods (as per interface above)
    ...

# creating a solver object
solver = Solver(name='cadical195', bootstrap_with=some_formula)

engine = MyPowerfulEngine(...)
solver.connect_propagator(engine)

# attached propagator wants to observe these variables
for var in range(some_variables):
    solver.observe(var)

...

```

A quick example

defining a new engine

```python
class MyPowerfulEngine(Propagator):
    # define all the methods (as per interface above)
    ...

# creating a solver object
solver = Solver(name='cadical195', bootstrap_with=some_formula)

engine = MyPowerfulEngine(...)
solver.connect_propagator(engine)

# attached propagator wants to observe these variables
for var in range(some_variables):
    solver.observe(var)

...
```

A quick example

defining a new engine

```python
class MyPowerfulEngine(Propagator):
    # define all the methods (as per interface above)
    ...

# creating a solver object
solver = Solver(name='cadical195', bootstrap_with=some_formula)

engine = MyPowerfulEngine(...)
solver.connect_propagator(engine)

# attached propagator wants to observe these variables
for var in range(some_variables):
    solver.observe(var)

...

```

connecting the engine to the solver

A quick example

defining a new engine

```
class MyPowerfulEngine(Propagator):
    # define all the methods (as per interface above)
    ...

# creating a solver object
solver = Solver(name='cadical195', bootstrap_with=some_formula)

engine = MyPowerfulEngine(...)
solver.connect_propagator(engine)

# attached propagator wants to observe these variables
for var in range(some_variables):
    solver.observe(var)

...
```

connecting the engine to the solver

solver will keep the engine posted on these

Experimenting with external engines

Experimental setup

- **CaDiCaL 1.9.5, augmented to support the engine interface**

Experimental setup

- **CaDiCaL 1.9.5, augmented to support the engine interface**
- **example engine `BooleanEngine`**
    - linear constraints (cardinality, pseudo-Boolean) + XOR constraints
    - can disable/re-enable itself on the fly

Experimental setup

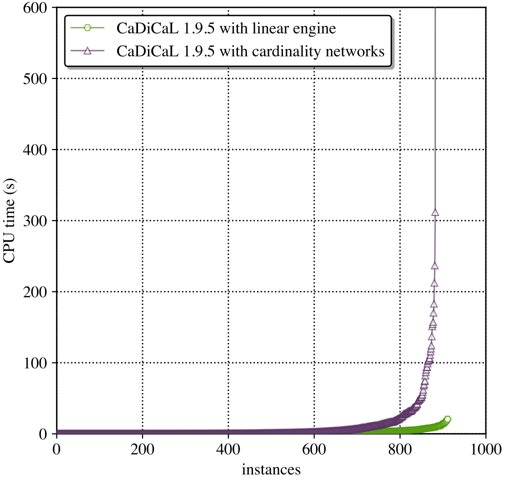
- **CaDiCaL 1.9.5, augmented to support the engine interface**
- **example engine `BooleanEngine`**
  - linear constraints (cardinality, pseudo-Boolean) + XOR constraints
  - can disable/re-enable itself on the fly
- **MacBook Pro**
  - 10-core Apple M1 Pro with 32GByte RAM
  - running macOS Sonoma 14.3.1
  - timeout 10 minutes

Experimental setup

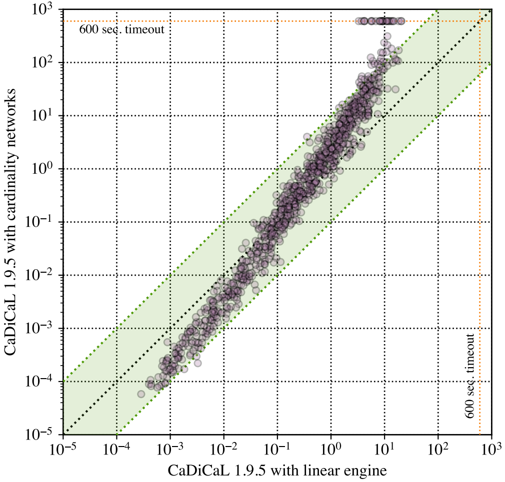
- **CaDiCaL 1.9.5, augmented to support the engine interface**
- **example engine `BooleanEngine`**
  - linear constraints (cardinality, pseudo-Boolean) + XOR constraints
  - can disable/re-enable itself on the fly
- **MacBook Pro**
  - 10-core Apple M1 Pro with 32GByte RAM
  - running macOS Sonoma 14.3.1
  - timeout 10 minutes
- **two benchmark families:**
  1. model enumeration for cardinality constraints
  2. abductive explanation of tree ensembles

Model enumeration

$\sum_{i=1}^{20} w_i \cdot l_i \leq v, \quad l_i \in \{x_i, \neg x_i\}$, and $w_i \in \{0, 1\}, v \in \{0, 1, \ldots, 20\}$



(a) Overall performance

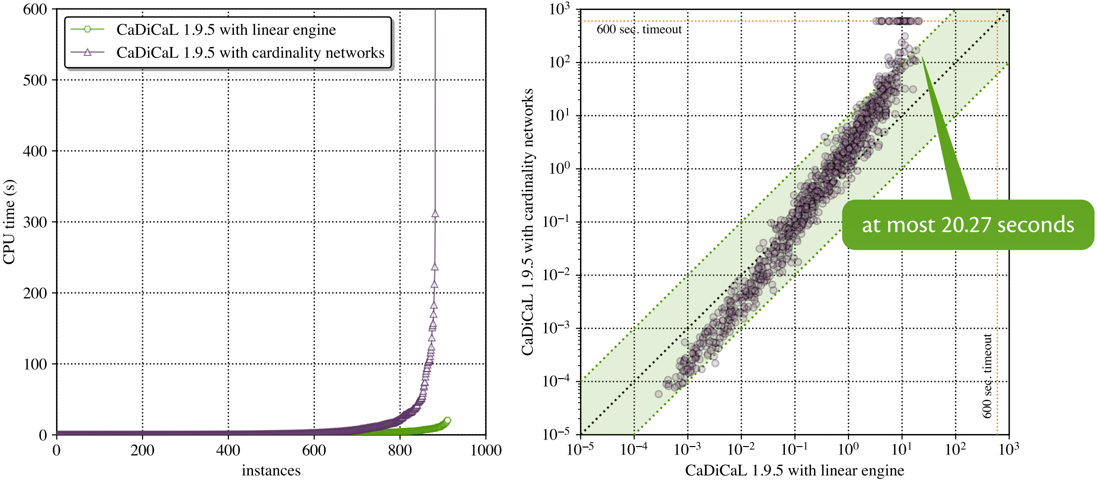


(b) Instance-by-instance comparison

1 to 1,044,905 models to enumerate

Model enumeration

$\sum_{i=1}^{20} w_i \cdot l_i \leq v, \quad l_i \in \{x_i, \neg x_i\}$, and $w_i \in \{0, 1\}, v \in \{0, 1, \ldots, 20\}$



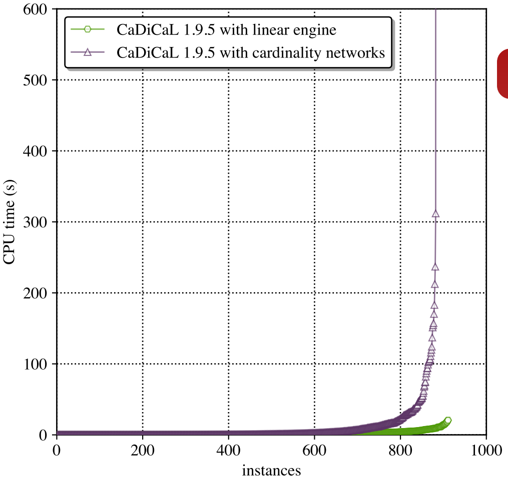
(a) Overall performance

(b) Instance-by-instance comparison

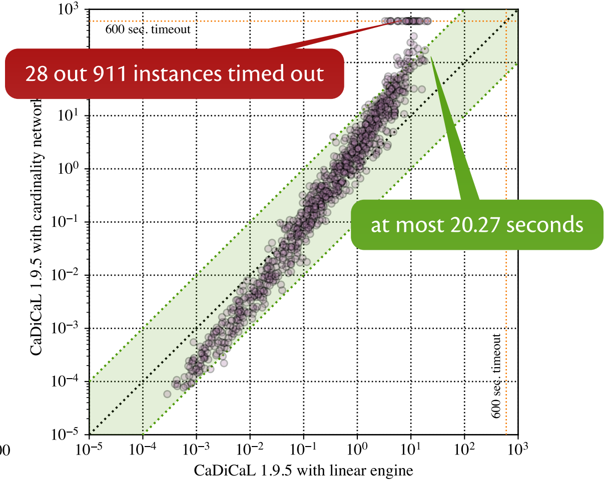
1 to 1,044,905 models to enumerate

Model enumeration

$\sum_{i=1}^{20} w_i \cdot l_i \leq v, \quad l_i \in \{x_i, \neg x_i\}$, and $w_i \in \{0, 1\}, v \in \{0, 1, \ldots, 20\}$



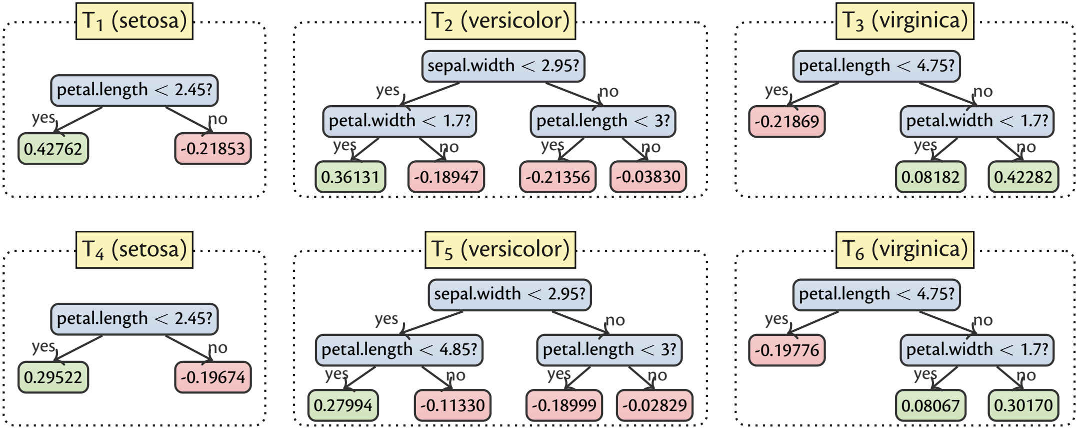
(a) Overall performance



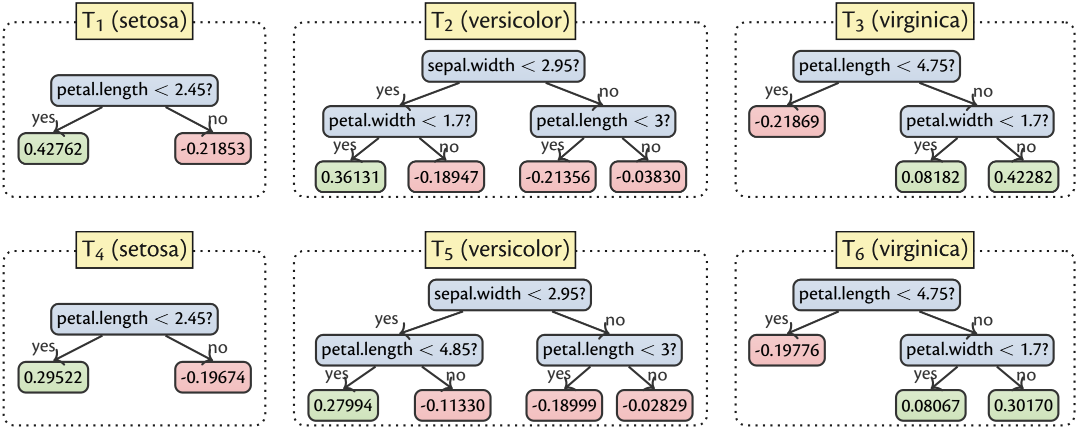
(b) Instance-by-instance comparison

1 to 1,044,905 models to enumerate

AXp example – tree ensembles

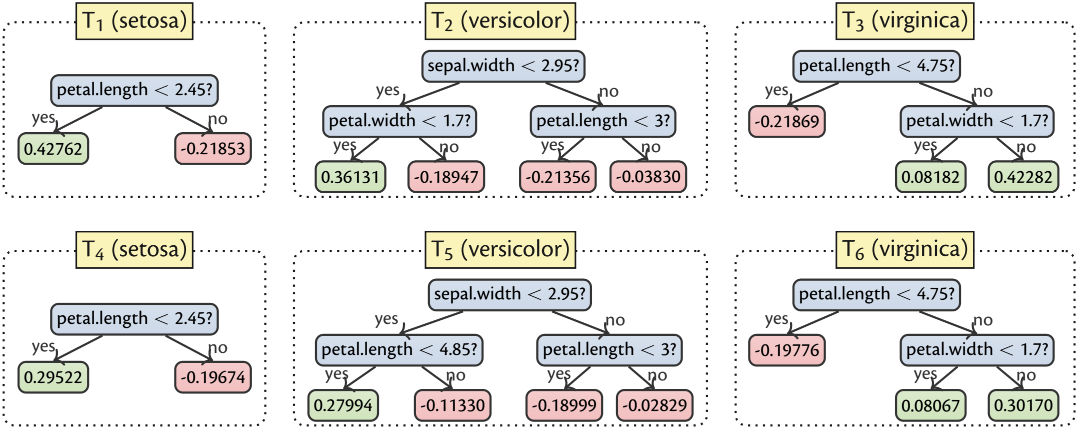


AXp example – tree ensembles



- $w(x, c) = \sum_{j \in \{0,\ldots,n-1\}} \mathcal{T}_{Kj+c}(x), \; c \in [K]$
- $\tau(x) = \arg\max_{c \in [K]} w(x, c)$

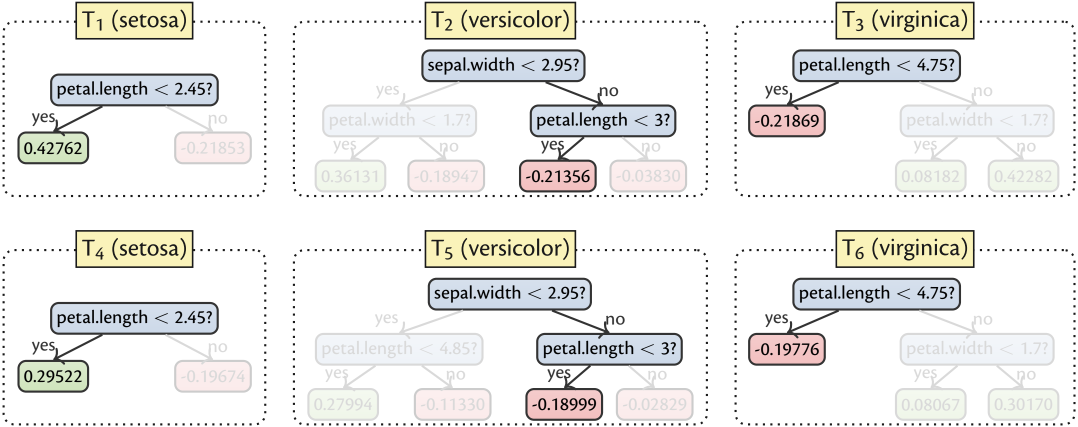
AXp example – tree ensembles



- $w(x, c) = \sum_{j \in \{0,\dots,n-1\}} \mathcal{T}_{Kj+c}(x), \; c \in [K]$
- $\tau(x) = \arg\max_{c \in [K]} w(x, c)$

$\forall (x \in \mathbb{F}). \bigwedge_{j \in \mathcal{X}} (x_j = v_j) \rightarrow (\tau(x) = c)$

AXp example – tree ensembles

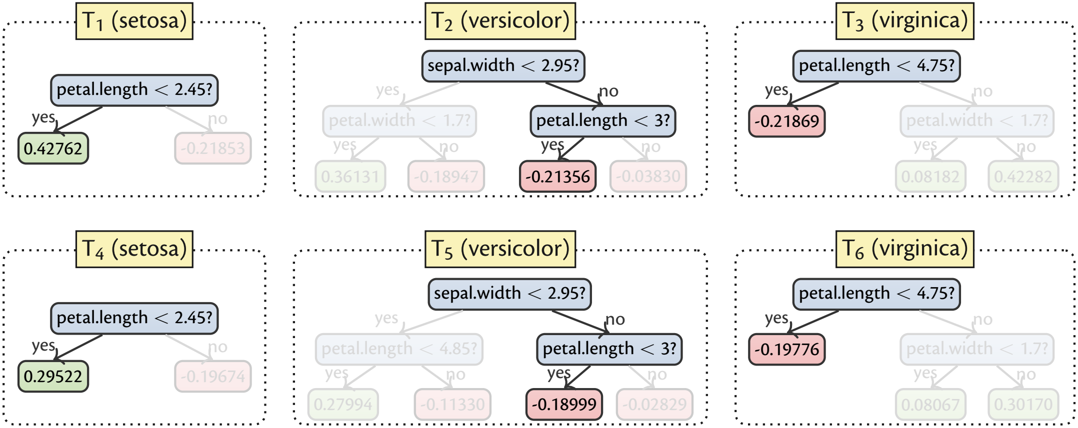


- $w(x, c) = \sum_{j \in \{0,\dots,n-1\}} \mathcal{T}_{Kj+c}(x), \; c \in [K]$
- $\tau(x) = \arg\max_{c \in [K]} w(x, c)$

$(\text{sepal.length} = 5.1) \wedge (\text{sepal.width} = 3.5) \wedge (\text{petal.length} = 1.4) \wedge (\text{petal.width} = 0.2)$

$\forall (x \in \mathbb{F}). \bigwedge_{j \in \mathcal{X}} (x_j = v_j) \rightarrow (\tau(x) = c)$

AXp example – tree ensembles

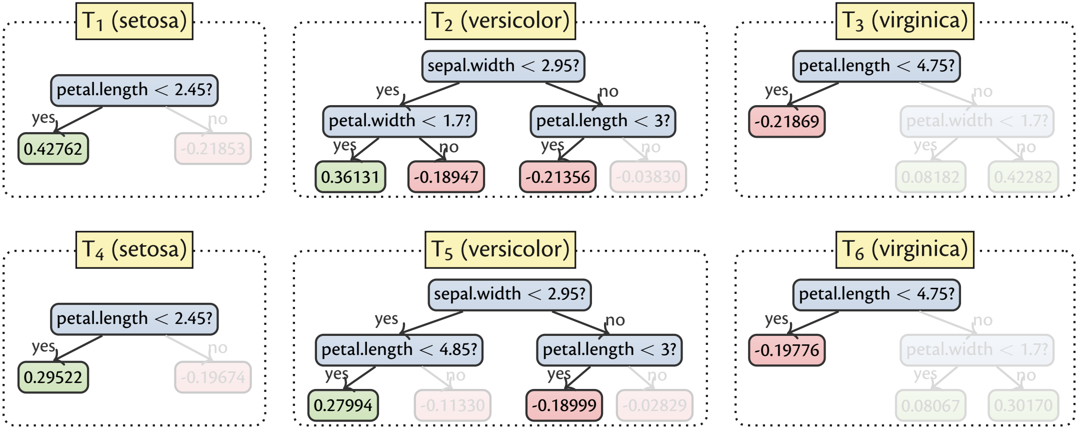


- $w(x, c) = \sum_{j \in \{0,\dots,n-1\}} \mathcal{T}_{Kj+c}(x),\ c \in [K]$
- $\tau(x) = \arg\max_{c \in [K]} w(x, c)$

$(\text{sepal.length} = 5.1) \wedge (\text{sepal.width} =$ $\mathcal{X} = \{(\text{petal.length} = 1.4)\}$ $(\text{petal.width} = 0.2)$

$\forall (x \in \mathbb{F}). \bigwedge_{j \in \mathcal{X}} (x_j = v_j) \rightarrow (\tau(x) = c)$

AXp example – tree ensembles

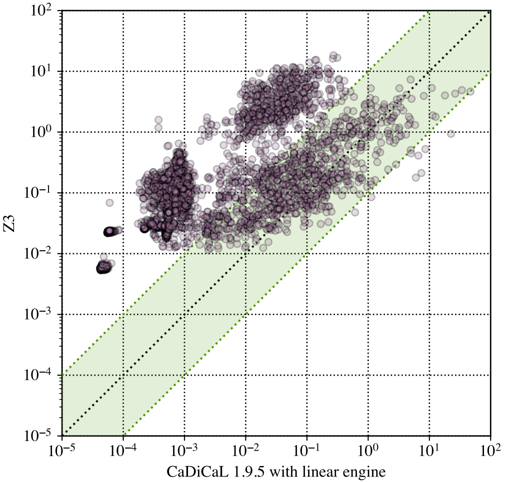


- $w(x, c) = \sum_{j \in \{0,\dots,n-1\}} \mathcal{T}_{Kj+c}(x), \ c \in [K]$
- $\tau(x) = \arg\max_{c \in [K]} w(x, c)$

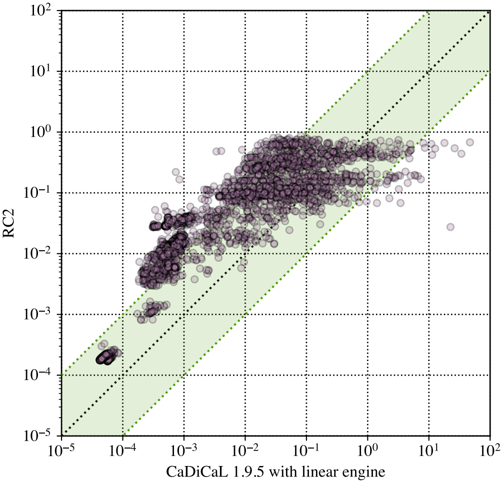
$(\text{sepal.length} = 5.1) \wedge (\text{sepal.width} =$ $\mathcal{X} = \{(\text{petal.length} = 1.4)\}$ $(\text{petal.width} = 0.2)$

$\forall(x \in \mathbb{F}). \bigwedge_{j \in \mathcal{X}} (x_j = v_j) \rightarrow (\tau(x) = c)$

Single explanation extraction

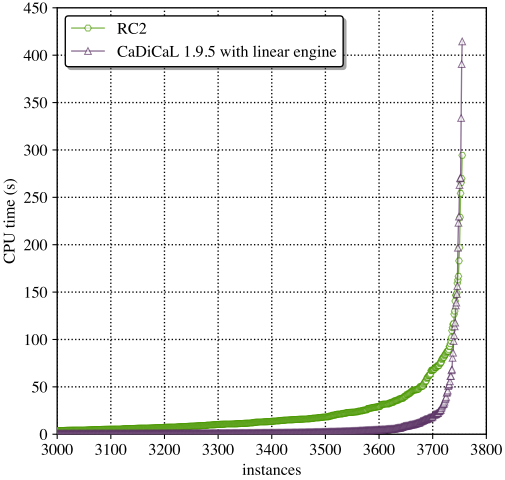


(a) Comparison against Z3

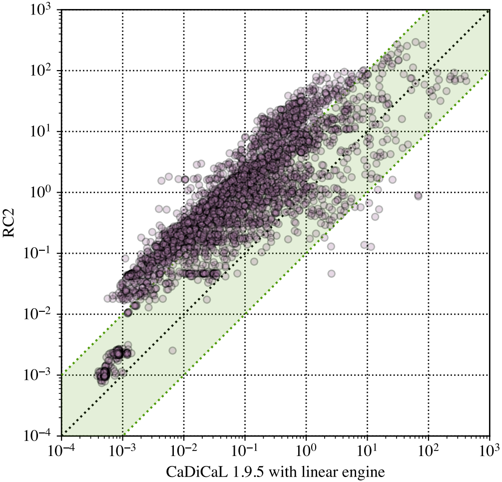


(b) Comparison against RC2

Enumerating 100 explanations



(a) Overall performance



(b) Instance-by-instance comparison

Installation & platforms

- `$ pip install python-sat[aiger,approxmc,cryptosat,pblib]`

Installation & platforms

- `$ pip install python-sat[aiger,approxmc,cryptosat,pblib]`
- **Linux, MacOS, Windows**

Installation & platforms

- `$ pip install python-sat[aiger,approxmc,cryptosat,pblib]`
- **Linux, MacOS, Windows**
- **no compilation needed**

Installation & platforms

- `$ pip install python-sat[aiger,approxmc,cryptosat,pblib]`
- **Linux, MacOS, Windows**
- **no compilation needed**
- **+ online REPL platforms, e.g.,** `repl.it`

Installation & platforms

- `$ pip install python-sat[aiger,approxmc,cryptosat,pblib]`
- **Linux, MacOS, Windows**
- **no compilation needed**
- **+ online REPL platforms, e.g., `repl.it`**
- **+ PySAT is a part of `Pyodide`**
  (Python scientific stack compiled to WebAssembly)

https://pysathq.github.io/