Modelling and Solving Problems with SAT

From circuits to MaxSAT

Alexey Ignatiev

ACP Summer School 2026

MONASH University

Modelling with SAT

Tseitin transformation of non-clausal formulas / circuits I

- **Logic connectives:** $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$

Tseitin transformation of non-clausal formulas / circuits I

- **Logic connectives:** $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$
- **Given a gate,** CNF formula encodes the **consistent** assignments to its inputs and output

Tseitin transformation of non-clausal formulas / circuits I

- **Logic connectives:** $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$
- **Given a gate,** CNF formula encodes the **consistent** assignments to its inputs and output
- Given $z = \mathrm{OP}(x, y)$,

Tseitin transformation of non-clausal formulas / circuits I

- **Logic connectives:** $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$
- **Given a gate,** CNF formula encodes the **consistent** assignments to its inputs and output
- Given $z = \mathrm{OP}(x, y)$, **represent in CNF**

$$z \leftrightarrow \mathbf{OP}(x, y)$$

Tseitin transformation of non-clausal formulas / circuits I

- **Logic connectives:** $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$
- **Given a gate,** CNF formula encodes the **consistent** assignments to its inputs and output
- Given $z = \mathrm{OP}(x, y)$, **represent in CNF**

$$z \leftrightarrow \mathbf{OP}(x, y)$$

- CNF formula for the circuit is the **conjunction** of CNF formula for each gate

Tseitin transformation of non-clausal formulas / circuits I

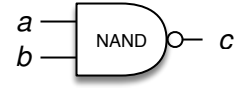
- **Logic connectives:** $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$
- **Given a gate,** CNF formula encodes the **consistent** assignments to its inputs and output
- Given $z = \mathrm{OP}(x, y)$, **represent in CNF**

$$z \leftrightarrow \mathbf{OP}(x, y)$$

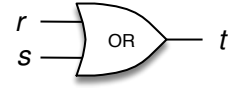
- CNF formula for the circuit is the **conjunction** of CNF formula for each gate

Example

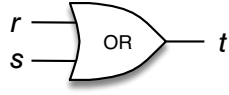
$$\mathcal{F}_c = (a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$



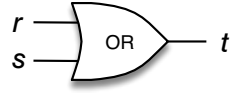
$$\mathcal{F}_t = (\neg r \vee t) \wedge (\neg s \vee t) \wedge (r \vee s \vee \neg t)$$



Tseitin transformation of non-clausal formulas / circuits II

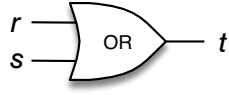


Tseitin transformation of non-clausal formulas / circuits II



$$t \leftrightarrow (r \vee s)$$

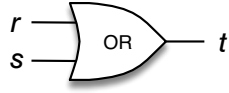
Tseitin transformation of non-clausal formulas / circuits II



$$t \leftrightarrow (r \vee s)$$

$$t \rightarrow (r \vee s) \quad \wedge \quad t \leftarrow (r \vee s)$$

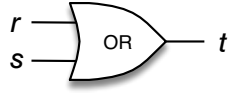
Tseitin transformation of non-clausal formulas / circuits II



$$t \leftrightarrow (r \vee s)$$

$$\begin{array}{lll} t \rightarrow (r \vee s) & \wedge & t \leftarrow (r \vee s) \\ (\neg t \vee r \vee s) & \wedge & t \vee \neg(r \vee s) \end{array}$$

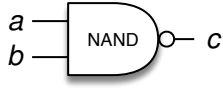
Tseitin transformation of non-clausal formulas / circuits II



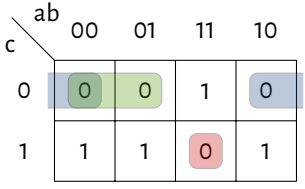
$$t \leftrightarrow (r \vee s)$$

$$\begin{array}{lcl} t \rightarrow (r \vee s) & \wedge & t \leftarrow (r \vee s) \\ (\neg t \vee r \vee s) & \wedge & t \vee \neg(r \vee s) \\ (\neg t \vee r \vee s) & \wedge & (t \vee \neg r) \wedge (r \vee \neg s) \end{array}$$

Tseitin transformation of non-clausal formulas / circuits III



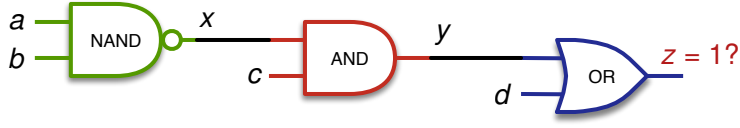
a	b	c	$\mathcal{F}_c$(a,b,c)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



$$\mathcal{F}_c = (a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

Tseitin transformation of non-clausal formulas / circuits IV

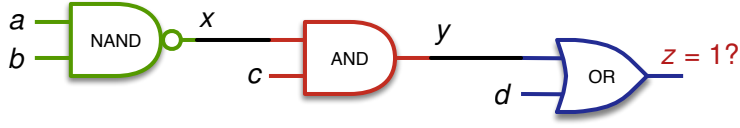
- **CNF formula for the circuit is the conjunction of the CNF formula for each gate**
  - Can specify ***objectives with additional clauses***



$$\mathcal{F} = (a \vee x) \wedge (b \vee x) \wedge (\neg a \vee \neg b \vee \neg x) \wedge$$
$$(x \vee \neg y) \wedge (c \vee \neg y) \wedge (\neg x \vee \neg c \vee y) \wedge$$
$$(\neg y \vee z) \wedge (\neg d \vee z) \wedge (y \vee d \vee \neg z) \wedge \qquad (z)$$

Tseitin transformation of non-clausal formulas / circuits IV

- **CNF formula for the circuit is the conjunction of the CNF formula for each gate**
  - Can specify ***objectives with additional clauses***

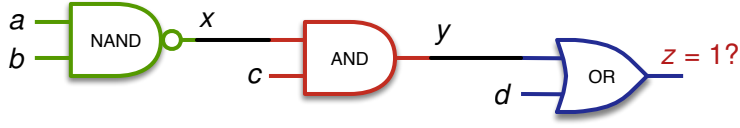


$$\mathcal{F} = (a \vee x) \wedge (b \vee x) \wedge (\neg a \vee \neg b \vee \neg x) \wedge$$
$$(x \vee \neg y) \wedge (c \vee \neg y) \wedge (\neg x \vee \neg c \vee y) \wedge$$
$$(\neg y \vee z) \wedge (\neg d \vee z) \wedge (y \vee d \vee \neg z) \wedge \qquad (z)$$

- **Note:** $z = d \vee (c \wedge (\neg(a \wedge b)))$

Tseitin transformation of non-clausal formulas / circuits IV

- **CNF formula for the circuit is the conjunction of the CNF formula for each gate**
  - Can specify ***objectives with additional clauses***



$$\mathcal{F} = (a \vee x) \wedge (b \vee x) \wedge (\neg a \vee \neg b \vee \neg x) \wedge$$
$$(x \vee \neg y) \wedge (c \vee \neg y) \wedge (\neg x \vee \neg c \vee y) \wedge$$
$$(\neg y \vee z) \wedge (\neg d \vee z) \wedge (y \vee d \vee \neg z) \wedge \qquad (z)$$

- **Note:** $z = d \vee (c \wedge (\neg(a \wedge b)))$
- Easy to do more structures: **ITEs**; **Adders**; etc.

Practical example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>>
>>> f = ((~x >> y) & z) | (~(~x >> y) | (x & z))
>>> f.clausify()
>>>
>>> list(f)[:6]  # first 6 clauses
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4]]
>>>
>>> list(f)[6:]  # last 4 clauses
[[3, -6], [4, -6], [6, -3, -4], [-3, 5, 6]]
>>>
>>>
>>> for id, obj in Formula._vpool['default'].id2obj.items():
...     print(f'{"+" if id > 0 else ""}{id}:\t{obj}')
...
+1:     x
-1:     ~x
+2:     y
+3:     ~x >> y
-3:     ~(~x >> y)
+4:     z
+5:     x & z
+6:     (~x >> y) & z
```

Practical example

$f \triangleq ((\neg x \rightarrow y) \wedge z) \vee (\neg(\neg x \rightarrow y) \vee (x \wedge z))$

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>>
>>> f = ((~x >> y) & z) | (~(~x >> y) | (x & z))
>>> f.clausify()
>>>
>>> list(f)[:6]  # first 6 clauses
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4]]
>>>
>>> list(f)[6:]  # last 4 clauses
[[3, -6], [4, -6], [6, -3, -4], [-3, 5, 6]]
>>>
>>>
>>> for id, obj in Formula._vpool['default'].id2obj.items():
...     print(f'{"+" if id > 0 else ""}{id}:\t{obj}')
...
+1:     x
-1:     ~x
+2:     y
+3:     ~x >> y
-3:     ~(~x >> y)
+4:     z
+5:     x & z
+6:     (~x >> y) & z
```

Practical example

$f \triangleq ((\neg x \rightarrow y) \wedge z) \vee (\neg(\neg x \rightarrow y) \vee (x \wedge z))$

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>>
>>> f = ((~x >> y) & z) | (~(~x >> y) | (x & z))
>>> f.clausify()
>>>
>>> list(f)[:6]  # first 6 clauses
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4]]
>>>
>>> list(f)[6:]  # last 4 clauses
[[3, -6], [4, -6], [6, -3, -4], [-3, 5, 6]]
>>>
>>>
>>> for id, obj in Formula._vpool['default'].id2obj.items():
...     print(f'{"+" if id > 0 else ""}{id}:\t{obj}')
...
+1:     x
-1:     ~x
+2:     y
+3:     ~x >> y
-3:     ~(~x >> y)
+4:     z
+5:     x & z
+6:     (~x >> y) & z
```

all the used literals and the formulas they represent

Practical example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>>
>>> f = ((~x >> y) & z) | (~(~x >> y) | (x & z))
>>> f.clausify()
>>>
>>> list(f)[:6]  # first 6 clauses
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4]]
>>>
>>> list(f)[6:]  # last 4 clauses
[[3, -6], [4, -6], [6, -3, -4], [-3, 5, 6]]
>>>
>>>
>>> for id, obj in Formula._vpool['default'].id2obj.items():
...     print(f'{"+" if id > 0 else ""}{id}:\t{obj}')
...
+1:     x
-1:     ~x
+2:     y
+3:     ~x >> y
-3:     ~(~x >> y)
+4:     z
+5:     x & z
+6:     (~x >> y) & z
```

$f \triangleq ((\neg x \rightarrow y) \wedge z) \vee (\neg(\neg x \rightarrow y) \vee (x \wedge z))$

encoding of $v_3 \triangleq (\neg x \rightarrow y)$

all the used literals and the formulas they represent

Practical example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>>
>>> f = ((~x >> y) & z) | (~(~x >> y) | (x & z))
>>> f.clausify()
>>>
>>> list(f)[:6]  # first 6 clauses
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4]]
>>>
>>> list(f)[6:]  # last 4 clauses
[[3, -6], [4, -6], [6, -3, -4], [-3, 5, 6]]
>>>
>>>
>>> for id, obj in Formula._vpool['default'].id2obj.items():
...     print(f'{"+" if id > 0 else ""}{id}:\t{obj}')
...
+1:     x
-1:     ~x
+2:     y
+3:     ~x >> y
-3:     ~(~x >> y)
+4:     z
+5:     x & z
+6:     (~x >> y) & z
```

$f \triangleq ((\neg x \rightarrow y) \wedge z) \vee (\neg(\neg x \rightarrow y) \vee (x \wedge z))$

encoding of $v_3 \triangleq (\neg x \rightarrow y)$

encoding of $v_5 \triangleq (x \wedge z)$

all the used literals and the formulas they represent

Practical example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>>
>>> f = ((~x >> y) & z) | (~(~x >> y) | (x & z))
>>> f.clausify()
>>>
>>> list(f)[:6]  # first 6 clauses
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4]]
>>>
>>> list(f)[6:]  # last 4 clauses
[[3, -6], [4, -6], [6, -3, -4], [-3, 5, 6]]
>>>
>>>
>>> for id, obj in f.auxvars[-1].id2obj.items():
...     print(f'{"+" if id > 0 else ""}{id}:\t{obj}')
...
+1:     x
-1:     ~x
+2:     y
+3:     ~x >> y
-3:     ~(~x >> y)
+4:     z
+5:     x & z
+6:     (~x >> y) & z
```

$f \triangleq ((\neg x \rightarrow y) \wedge z) \vee (\neg(\neg x \rightarrow y) \vee (x \wedge z))$

encoding of $v_3 \triangleq (\neg x \rightarrow y)$

encoding of $v_5 \triangleq (x \wedge z)$

encoding of $v_6 \triangleq (v_3 \wedge z)$

all the used literals and the formulas they represent

Practical example

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>>
>>> f = ((~x >> y) & z) | (~(~x >> y) | (x & z))
>>> f.clausify()
>>>
>>> list(f)[:6]  # first 6 clauses
[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4]]
>>>
>>> list(f)[6:]  # last 4 clauses
[[3, -6], [4, -6], [6, -3, -4], [-3, 5, 6]]
>>>
>>>
>>> for id, obj in f.auxvars[-1].id2obj.items():
...     print(f'{"+" if id > 0 else ""}{id}:\t{obj}')
...
+1:     x
-1:     ~x
+2:     y
+3:     ~x >> y
-3:     ~(~x >> y)
+4:     z
+5:     x & z
+6:     (~x >> y) & z
```

$f \triangleq ((\neg x \rightarrow y) \wedge z) \vee (\neg(\neg x \rightarrow y) \vee (x \wedge z))$

encoding of $v_3 \triangleq (\neg x \rightarrow y)$

encoding of $v_5 \triangleq (x \wedge z)$

enforcing f to be true

encoding of $v_6 \triangleq (v_3 \wedge z)$

all the used literals and the formulas they represent

exercise time

Cardinality and PB constraints

- **Cardinality constraints:** $\sum_{j=1}^{n} x_j \leqslant k$

- **Cardinality constraints:** $\sum_{j=1}^{n} x_j \leqslant k$
  - **Variables x_j are Boolean**

- **Cardinality constraints:** $\sum_{j=1}^{n} x_j \leqslant k$
  - **Variables x_j are Boolean**
  - **General form:** $\sum_{j=1}^{n} x_j \bowtie k$, with $\bowtie \in \{<, \leqslant, =, \geqslant, >\}$

- **Cardinality constraints:** $\sum_{j=1}^{n} x_j \leqslant k$
  - **Variables x_j are Boolean**
  - **General form:** $\sum_{j=1}^{n} x_j \bowtie k$, with $\bowtie \in \{<, \leqslant, =, \geqslant, >\}$
- **Pseudo-Boolean constraints:**

$$\sum_{j=1}^{n} a_j x_j \bowtie k, \text{ with } \bowtie \in \{<, \leqslant, =, \geqslant, >\}$$

Equals1, AtLeast1 & AtMost1 constraints

- $\sum_{j=1}^{n} x_j = 1$ is encoded as $(\sum_{j=1}^{n} x_j \leqslant 1) \wedge (\sum_{j=1}^{n} x_j \geqslant 1)$

Equals1, AtLeast1 & AtMost1 constraints

- $\sum_{j=1}^{n} x_j = 1$ is encoded as $(\sum_{j=1}^{n} x_j \leqslant 1) \wedge (\sum_{j=1}^{n} x_j \geqslant 1)$
- $\sum_{j=1}^{n} x_j \geqslant 1$ is encoded as $(x_1 \vee x_2 \vee \ldots \vee x_n)$

Equals1, AtLeast1 & AtMost1 constraints

- $\sum_{j=1}^{n} x_j = 1$ is encoded as $(\sum_{j=1}^{n} x_j \leqslant 1) \wedge (\sum_{j=1}^{n} x_j \geqslant 1)$
- $\sum_{j=1}^{n} x_j \geqslant 1$ is encoded as $(x_1 \vee x_2 \vee \ldots \vee x_n)$
- $\sum_{j=1}^{n} x_j \leqslant 1$ is encoded through:
  - **Pairwise encoding**
    - Clauses: $\mathcal{O}(n^2)$; No auxiliary variables
  - **Sequential counter**
    - Clauses: $\mathcal{O}(n)$; Auxiliary variables: $\mathcal{O}(n)$
  - **Bitwise encoding**
    - Clauses: $\mathcal{O}(n \log n)$; Auxiliary variables: $\mathcal{O}(\log n)$
  - ...

plenty of various encodings exist!

Pairwise encoding for AtMost1

- **How to (propositionally) encode AtMost1 constraint** $a + b + c + d \leqslant 1$**?**

Pairwise encoding for AtMost1

- **How to (propositionally) encode AtMost1 constraint $a + b + c + d \leqslant 1$?**

$$
\begin{aligned}
a \to \neg b \wedge \neg c \wedge \neg d &\implies (\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee \neg d) \\
b \to \neg c \wedge \neg d \wedge \neg a &\implies (\neg b \vee \neg c) \wedge (\neg b \vee \neg d) \wedge (\neg b \vee \neg a) \\
c \to \neg d \wedge \neg a \wedge \neg b &\implies (\neg c \vee \neg d) \wedge (\neg c \vee \neg a) \wedge (\neg c \vee \neg b) \\
d \to \neg a \wedge \neg b \wedge \neg c &\implies (\neg d \vee \neg a) \wedge (\neg d \vee \neg b) \wedge (\neg d \vee \neg c)
\end{aligned}
$$

  - **Encoded as:** $(\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee \neg d) \wedge (\neg b \vee \neg c) \wedge (\neg b \vee \neg d) \wedge (\neg c \vee \neg d)$

Pairwise encoding for AtMost1

- **How to (propositionally) encode AtMost1 constraint $a + b + c + d \leqslant 1$?**

$$
\begin{aligned}
a \to \neg b \wedge \neg c \wedge \neg d \quad &\Longrightarrow \quad (\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee \neg d) \\
b \to \neg c \wedge \neg d \wedge \neg a \quad &\Longrightarrow \quad (\neg b \vee \neg c) \wedge (\neg b \vee \neg d) \wedge (\neg b \vee \neg a) \\
c \to \neg d \wedge \neg a \wedge \neg b \quad &\Longrightarrow \quad (\neg c \vee \neg d) \wedge (\neg c \vee \neg a) \wedge (\neg c \vee \neg b) \\
d \to \neg a \wedge \neg b \wedge \neg c \quad &\Longrightarrow \quad (\neg d \vee \neg a) \wedge (\neg d \vee \neg b) \wedge (\neg d \vee \neg c)
\end{aligned}
$$

  - **Encoded as:** $(\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg a \vee \neg d) \wedge (\neg b \vee \neg c) \wedge (\neg b \vee \neg d) \wedge (\neg c \vee \neg d)$

- **With n variables, number of clauses becomes $\frac{n(n-1)}{2}$**
  - But no additional variables

Generalized pairwise encoding

- **General form:** $\sum_{j=1}^{n} x_j \leqslant k$
- **Any combination of $k+1$ true variables is disallowed**

Generalized pairwise encoding

- **General form:** $\sum_{j=1}^{n} x_j \leqslant k$
- **Any combination of $k+1$ true variables is disallowed**

- **Example:** $a + b + c + d \leqslant 2$

Generalized pairwise encoding

- **General form:** $\sum_{j=1}^{n} x_j \leqslant k$
- **Any combination of $k+1$ true variables is disallowed**

- **Example:** $a + b + c + d \leqslant 2$

$$
\begin{aligned}
a \wedge b \rightarrow \neg c &\implies (\neg a \vee \neg b \vee \neg c) \\
a \wedge b \rightarrow \neg d &\implies (\neg a \vee \neg b \vee \neg d) \\
a \wedge c \rightarrow \neg d &\implies (\neg a \vee \neg c \vee \neg d) \\
b \wedge c \rightarrow \neg d &\implies (\neg b \vee \neg c \vee \neg d)
\end{aligned}
$$

  - Encoded as: $(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg c \vee \neg d) \wedge (\neg b \vee \neg c \vee \neg d)$

Generalized pairwise encoding

- **General form:** $\sum_{j=1}^{n} x_j \leqslant k$
- **Any combination of $k+1$ true variables is disallowed**

- **Example:** $a + b + c + d \leqslant 2$

$$
\begin{aligned}
a \wedge b \rightarrow \neg c &\implies (\neg a \vee \neg b \vee \neg c) \\
a \wedge b \rightarrow \neg d &\implies (\neg a \vee \neg b \vee \neg d) \\
a \wedge c \rightarrow \neg d &\implies (\neg a \vee \neg c \vee \neg d) \\
b \wedge c \rightarrow \neg d &\implies (\neg b \vee \neg c \vee \neg d)
\end{aligned}
$$

  - Encoded as: $(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg c \vee \neg d) \wedge (\neg b \vee \neg c \vee \neg d)$

- **In general, number of clauses is** $\binom{n}{k+1}$
  - Recall: for AtMost1 (i.e. for $k = 1$), number of clauses is: $\frac{n(n-1)}{2}$

Sequential counter encoding

- **Encode** $\sum_{j=1}^{n} x_j \leqslant 1$ **with sequential counter:**

$$(\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}))$$

Sequential counter encoding

- **Encode $\sum_{j=1}^{n} x_j \leqslant 1$ with sequential counter:**

$$(\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}))$$

  - **If some $x_j = 1$, then all s_i variables must be assigned**
    - $s_i = 1$ for $i \geqslant j$, and so $x_i = 0$ for $i > j$
    - $s_i = 0$ for $i < j$, and so $x_i = 0$ for $i < j$
    - Thus, all other x_i variables must take value 0

Sequential counter encoding

- **Encode $\sum_{j=1}^{n} x_j \leqslant 1$ with sequential counter:**

$$(\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}))$$

  - **If some $x_j = 1$, then all s_i variables must be assigned**
    - $s_i = 1$ for $i \geqslant j$, and so $x_i = 0$ for $i > j$
    - $s_i = 0$ for $i < j$, and so $x_i = 0$ for $i < j$
    - Thus, all other x_i variables must take value 0

  - **If all $x_j = 0$, can find consistent assignment to s_i variables**

Sequential counter encoding

- **Encode $\sum_{j=1}^{n} x_j \leqslant 1$ with sequential counter:**

$$(\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}))$$

  - **If some $x_j = 1$, then all s_i variables must be assigned**
    - $s_i = 1$ for $i \geqslant j$, and so $x_i = 0$ for $i > j$
    - $s_i = 0$ for $i < j$, and so $x_i = 0$ for $i < j$
    - Thus, all other x_i variables must take value 0
  - **If all $x_j = 0$, can find consistent assignment to s_i variables**
  - **$\mathcal{O}(n)$ clauses and $\mathcal{O}(n)$ auxiliary variables**

Sequential counter encoding – example

$$(\neg x_1 \vee s_1) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})) \wedge$$
$$(\neg x_n \vee \neg s_{n-1})$$

Example

- $x_1 + x_2 + x_3 + x_4 \leqslant 1$

Sequential counter encoding – example

$$(\neg x_1 \vee s_1) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})) \wedge$$
$$(\neg x_n \vee \neg s_{n-1})$$

Example

- $x_1 + x_2 + x_3 + x_4 \leqslant 1$
- $s_1, \ldots, s_3$ – sequential counter variables

Sequential counter encoding – example

$$(\neg x_1 \vee s_1) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})) \wedge$$
$$(\neg x_n \vee \neg s_{n-1})$$

Example

- $x_1 + x_2 + x_3 + x_4 \leqslant 1$
- $s_1, \ldots, s_3$ – sequential counter variables
- $i = 1$: $(\neg x_1 \vee s_1) \wedge$

Sequential counter encoding – example

$$(\neg x_1 \vee s_1) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})) \wedge$$
$$(\neg x_n \vee \neg s_{n-1})$$

Example

- $x_1 + x_2 + x_3 + x_4 \leqslant 1$
- $s_1, \ldots, s_3$ – sequential counter variables
- $i = 1$: $(\neg x_1 \vee s_1) \wedge$
- $i = 2$: $(\neg x_2 \vee s_2) \wedge (\neg s_1 \vee s_2) \wedge (\neg x_2 \vee \neg s_1) \wedge$

Sequential counter encoding – example

$$(\neg x_1 \vee s_1) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})) \wedge$$
$$(\neg x_n \vee \neg s_{n-1})$$

Example

- $x_1 + x_2 + x_3 + x_4 \leqslant 1$
- $s_1, \ldots, s_3$ – sequential counter variables
- $i = 1$: $(\neg x_1 \vee s_1) \wedge$
- $i = 2$: $(\neg x_2 \vee s_2) \wedge (\neg s_1 \vee s_2) \wedge (\neg x_2 \vee \neg s_1) \wedge$
- $i = 3$: $(\neg x_3 \vee s_3) \wedge (\neg s_2 \vee s_3) \wedge (\neg x_3 \vee \neg s_2) \wedge$

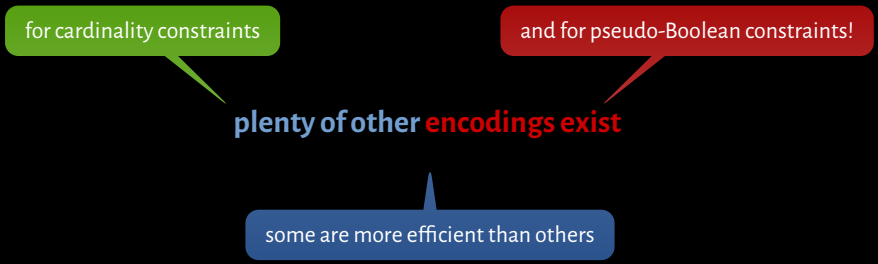
Sequential counter encoding – example

$$(\neg x_1 \vee s_1) \wedge$$
$$\bigwedge_{1<i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})) \wedge$$
$$(\neg x_n \vee \neg s_{n-1})$$

Example

- $x_1 + x_2 + x_3 + x_4 \leqslant 1$
- $s_1, \ldots, s_3$ – sequential counter variables
- $i = 1$: $(\neg x_1 \vee s_1) \wedge$
- $i = 2$: $(\neg x_2 \vee s_2) \wedge (\neg s_1 \vee s_2) \wedge (\neg x_2 \vee \neg s_1) \wedge$
- $i = 3$: $(\neg x_3 \vee s_3) \wedge (\neg s_2 \vee s_3) \wedge (\neg x_3 \vee \neg s_2) \wedge$
- $i = 4$: $(\neg x_4 \vee \neg s_3)$

plenty of other encodings exist


for cardinality constraints
and for pseudo-Boolean constraints!
plenty of other encodings exist
some are more efficient than others

Example: encoding “efficiency”

```
>>> from pysat.card import *
>>>
>>> encodings = ['bitwise', 'cardnetwrk', 'kmtotalizer', 'ladder',
                 'mtotalizer', 'pairwise', 'seqcounter', 'sortnetwrk', 'totalizer']
>>>
>>> for enc in encodings:
...     cnf = CardEnc.atmost(lits=range(1, 101), bound=1, encoding=getattr(EncType, enc))
...     print(f'encoding: {enc};\tvars: {cnf.nv};\tclauses: {len(cnf.clauses)}')
...
encoding: bitwise vars: 107     clauses: 700
encoding: cardnetwrk vars: 494     clauses: 592
encoding: kmtotalizer vars: 298     clauses: 396
encoding: ladder vars: 201     clauses: 400
encoding: mtotalizer vars: 558     clauses: 1800
encoding: pairwise vars: 100     clauses: 4950
encoding: seqcounter vars: 199     clauses: 296
encoding: sortnetwrk vars: 3043     clauses: 4415
encoding: totalizer vars: 772     clauses: 5623
```

affects SAT solver's performance!

Available cardinality encodings (`pysat.card` module)

Name	Type
pairwise	AtMost1
bitwise	AtMost1
ladder	AtMost1
sequential counter	AtMostk
sorting network	AtMostk
cardinality network	AtMostk
totalizer	AtMostk
mtotalizer	AtMostk
kmtotalizer	AtMostk

- Also AtLeastK and EqualsK constraints
- **URL**: `https://pysathq.github.io/docs/html/api/card.html`

`pysat.card` module

- `atleast()`, `atmost()`, and `equals()`

`pysat.card` module

- `atleast()`, `atmost()`, and `equals()`
- `ITotalizer` — *iterative totalizer*

`pysat.card` module

- `atleast()`, `atmost()`, and `equals()`
- `ITotalizer` — *iterative totalizer*

```
>>> from pysat.card import EncType, CardEnc
>>>
>>> cnf = CardEnc.atmost(lits=[1, 2, 3], bound=1, encoding=EncType.pairwise)
>>> print(cnf.clauses)
[[-1, -2], [-1, -3], [-2, -3]]
>>>
>>> cnf = CardEnc.equals(lits=[1, 2, 3], bound=1, encoding=EncType.pairwise)
>>> print(cnf.clauses)
[[1, 2, 3], [-1, -2], [-1, -3], [-2, -3]]
```

pysat.card—ITotalizer

- encode $\sum x_i \leqslant K$ constraints

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`
- increase right-hand side with `increase()`

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`
- increase right-hand side with `increase()`
- *merge* with another totalizer, with `merge_with()`

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`
- increase right-hand side with `increase()`
- *merge* with another totalizer, with `merge_with()`
- set bound incrementally

pysat.card—ITotalizer

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`
- increase right-hand side with `increase()`
- *merge* with another totalizer, with `merge_with()`
- set bound incrementally

```
>>> from pysat.card import ITotalizer
>>>
>>> t = ITotalizer(lits=[1, 2], ubound=1)
>>> print(t.cnf.clauses)
[[-2, 3], [-1, 3], [-1, -2, 4]]
>>> print(t.rhs)
[3, 4]
>>>
>>> t.extend(lits=[5], ubound=2)  # adding one more literal and increasing the bound
>>> print(t.cnf.clauses)
[[-2, 3], [-1, 3], [-1, -2, 4], [-5, 6], [-3, 6], [-4, 7], [-3, -5, 7], [-4, -5, 8]]
>>>
>>> print(t.cnf.clauses[-t.nof_new:])  # these are newly added clauses
[[-5, 6], [-3, 6], [-4, 7], [-3, -5, 7], [-4, -5, 8]]
>>>
>>> print(t.rhs)
[6, 7, 8]  # these literals represent bounds 0, 1, and 2
>>> t.delete()
```

exercise time

Incremental SAT and assumptions

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

- Solvers can either be used incrementally or non-incrementally

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

- Solvers can either be used incrementally or non-incrementally
- Tools can use multiple solvers, e.g. for hitting set dualization or CEGAR-based QBF solving

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

- Solvers can either be used incrementally or non-incrementally
- Tools can use multiple solvers, e.g. for hitting set dualization or CEGAR-based QBF solving
- Portfolio-like use of SAT oracles is possible

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

- Solvers can either be used incrementally or non-incrementally
- Tools can use multiple solvers, e.g. for hitting set dualization or CEGAR-based QBF solving
- Portfolio-like use of SAT oracles is possible
- **URL**: `https://pysathq.github.io/docs/html/api/solvers.html`

`pysat.solvers` module (basic interface)

- `s = Solver()&s.delete()`

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment
- `s.solve()` & `s.solve_limited()`

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment
- `s.solve()` & `s.solve_limited()`
- `s.add_clause()` & `s.append_formula()`

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment
- `s.solve()` & `s.solve_limited()`
- `s.add_clause()` & `s.append_formula()`
- `s.get_model()` & `s.get_core()` & `get_proof()`

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment
- `s.solve()` & `s.solve_limited()`
- `s.add_clause()` & `s.append_formula()`
- `s.get_model()` & `s.get_core()` & `get_proof()`

```
>>> from pysat.solvers import Solver, Minisat22
>>>
>>> s = Solver(name='g4')  # accessing Glucose4 by name
>>> s.add_clause([-1, 2])
>>> s.add_clause([-1, -2])
>>> s.solve()
True
>>> print(s.get_model())
[-1, -2]
>>> s.delete()
>>>
>>> with Minisat22(bootstrap_with=[[-1, 2], [-1, -2]]) as m:
...     print(m.solve())
...     print(m.get_model())
True
[-1, -2]
```

`pysat.solvers` — incremental SAT

- **SAT solver is used as a witness producing oracle**

- SAT solver is used as a witness producing oracle
  - satisfiable formula ➡ return a model

- SAT solver is used as a witness producing oracle
  - satisfiable formula ➡ return a model
  - unsatisfiable formula ⇨ return an unsatisfiable core

- SAT solver is used as a witness producing oracle
  - satisfiable formula ➧ return a model
  - unsatisfiable formula ⇨ return an unsatisfiable core

```
>>> from pysat.formula import CNF, IDPool
>>> from pysat.solvers import Solver

>>> vpool = IDPool()
>>> v = lambda obj: vpool.id(obj)

>>> cnf = CNF()
>>> cnf.extend([[-v('a'), v('b')], [-v('a'), v('c')]])
>>> cnf.extend([[v('a'), -v('s1')], [-v('b'), -v('c'), -v('s2')], [v('a'), -v('c'), -v('s3')], [v('a'), -v('b'), -v('s4')]])

>>> with Solver(bootstrap_with=cnf) as s:
...     if s.solve(assumptions=[v('s1'), -v('s2'), -v('s3'), v('s4')]) == True:
...         print('model:', ['{0}{1}'.format('' if v > 0 else '-', vpool.obj(abs(v))) for v in s.get_model()])
...
...     if s.solve(assumptions=[v('s1'), v('s2'), v('s3'), v('s4')]) == False:
...         print('ucore:', ['{0}{1}'.format('' if v > 0 else '-', vpool.obj(abs(v))) for v in s.get_core()])
...
model: ['a', 'b', 'c', 's1', '-s2', '-s3', 's4']
ucore: ['s2', 's1']
```

pysat.solvers — solve_limited()

```
>>> from pysat.solvers import MinisatGH
>>> from pysat.examples.genhard import Parity
>>>
>>> cnf = Parity(size=10)  # too hard for a SAT solver
>>> m = MinisatGH(bootstrap_with=cnf.clauses)
>>>
>>> m.prop_budget(100000)  # doing at most 100000 propagations
>>> print(m.solve_limited())  # making a limited oracle call
None
>>> m.delete()
```

pysat.solvers — solve_limited() and interrupt()

```
>>> from pysat.solvers import MinisatGH
>>> from pysat.examples.genhard import PHP
>>> from threading import Timer
>>>
>>> cnf = PHP(nof_holes=20)  # PHP20 is too hard for a SAT solver
>>> m = MinisatGH(bootstrap_with=cnf)
>>>
>>> def interrupt(s):
>>>     s.interrupt()
>>>
>>> timer = Timer(10, interrupt, [m])
>>> timer.start()
>>>
>>> print(m.solve_limited(expect_interrupt=True))
None
>>> m.delete()
```

pysat.solvers — propagate()

```
>>> from pysat.solvers import Glucose3
>>> from pysat.card import *
>>>
>>> cnf = CardEnc.atmost(lits=range(1, 6), bound=1, encoding=EncType.pairwise)
>>> g = Glucose3(bootstrap_with=cnf.clauses)
>>>
>>> g.propagate(assumptions=[1])
(True, [1, -2, -3, -4, -5])
>>>
>>> g.add_clause([2])
>>> g.propagate(assumptions=[1])
(False, [])
>>>
>>> g.delete()
```

- **+ lots of other cool stuff!**

  https://pysathq.github.io/docs/html/api/solvers.html

MUSes and MCSes

Example: naive (deletion) MUS extraction

Input : Set $\mathcal{F}$
Output: Minimal subset $\mathcal{M}$
begin
  $\mathcal{M} \leftarrow \mathcal{F}$
  foreach $c \in \mathcal{M}$:
    if $\neg\mathsf{SAT}(\mathcal{M} \setminus \{c\})$:
      $\mathcal{M} \leftarrow \mathcal{M} \setminus \{c\}$ // If $\neg\mathsf{SAT}(\mathcal{M} \setminus \{c\})$, then $c \notin$ **MUS**
  return $\mathcal{M}$ // Final $\mathcal{M}$ is MUS
end

- Number of predicate tests: $\mathcal{O}(m)$

[CD91, BDTW93]

Naive MUS extraction I

```
def main():
    cnf = CNF(from_file=argv[1])   # create a CNF object from file
    (rnv, assumps) = add_assumps(cnf)

    with Solver(name='cadical3', bootstrap_with=cnf) as oracle:

        mus = find_mus(assumps, oracle)
        mus = [ref - rnv for ref in mus]
        print('MUS: ', mus)

if __name__== '__main__':
    main()
```

Naive MUS extraction II

```python
def add_assumps(cnf):
    rnv = topv = cnf.nv
    assumps = []                        # list of assumptions to use
    for i in range(len(cnf.clauses)):
        topv += 1
        assumps.append(topv)            # register literal
        cnf.clauses[i].append(-topv)    # extend clause with literal
    cnf.nv = cnf.nv + len(assumps)      # update # of vars
    return rnv, assumps
```

```python
def main():
    cnf = CNF(from_file=argv[1])   # create a CNF object from file
    (rnv, assumps) = add_assumps(cnf)

    with Solver(name='cadical3', bootstrap_with=cnf) as oracle:

        mus = find_mus(assumps, oracle)
        mus = [ref - rnv for ref in mus]
        print('MUS: ', mus)

if __name__== '__main__':
    main()
```

Naive MUS extraction III

```
from sys import argv

from pysat.formula import CNF
from pysat.solvers import Solver


def find_mus(assumps, oracle):
    i = 0
    while i < len(assumps):
        ts = assumps[:i] + assumps[(i+1):]
        if not oracle.solve(assumptions=ts):
            assumps = ts
        else:
            i += 1
    return assumps
```

exercise time

From SAT to MaxSAT

Propositional Satisfiability (SAT)

Is there an assignment of values to variables that makes a given Boolean formula true?

Canonical NP-complete problem!

Propositional Satisfiability (SAT)

Is there an assignment of values to variables that makes a given Boolean formula true?

Canonical NP-complete problem!

Propositional Satisfiability (SAT)

Is there an assignment of values to variables that makes a given Boolean formula true?

Example:

· $F = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$

Canonical NP-complete problem!

Propositional Satisfiability (SAT)

Is there an assignment of values to variables that makes a given Boolean formula true?

Example:

- $F = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- $(x_1 = \text{true}, x_2 = \text{false})$ satisfies F

Canonical NP-complete problem!

Propositional Satisfiability (SAT)

Is there an assignment of values to variables that makes a given Boolean formula true?

Example:

- $F = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- $(x_1 = \text{true}, x_2 = \text{false})$ satisfies F

What if the formula is unsatisfiable?

- e.g. $F' = F \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$

Canonical NP-complete problem!

Propositional Satisfiability (SAT)

Is there an assignment of values to variables that makes a given Boolean formula true?

Example:

- $F = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- $(x_1 = \text{true}, x_2 = \text{false})$ satisfies F

What if the formula is unsatisfiable?

- e.g. $F' = F \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$

Maximum Satisfiability (MaxSAT)

Optimisation variant of SAT

*What is the **maximum number** of clauses that **can be satisfied** in a given formula?*

General MaxSAT: hard and soft clauses

- **Hard clauses** – **must be satisfied**
- **Soft clauses** – we **would like to satisfy**, *if possible*

General MaxSAT: hard and soft clauses

- **Hard clauses** – **must be satisfied**
- **Soft clauses** – we **would like to satisfy**, *if possible*
  - Associate a **cost** with *falsifying* each soft clause

General MaxSAT: hard and soft clauses

- **Hard clauses** – **must be satisfied**
- **Soft clauses** – we **would like to satisfy**, *if possible*
  - Associate a **cost** with *falsifying* each soft clause
  - For a hard clause, the cost can be viewed as ∞

General MaxSAT: hard and soft clauses

- **Hard clauses** – **must be satisfied**
- **Soft clauses** – we **would like to satisfy**, *if possible*
  - Associate a **cost** with *falsifying* each soft clause
  - For a hard clause, the cost can be viewed as ∞

Problem: find an assignment that satisfies all hard clauses and maximizes the sum of weights of satisfied soft clauses.

General MaxSAT: hard and soft clauses

- **Hard clauses** – **must be satisfied**
- **Soft clauses** – we **would like to satisfy**, *if possible*
  - Associate a **cost** with *falsifying* each soft clause
  - For a hard clause, the cost can be viewed as ∞

Problem: find an assignment that satisfies all hard clauses and maximizes the sum of weights of satisfied soft clauses.

- **Alternatively, can be seen a typical optimisation problem:**

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \varphi \end{aligned}$$

General MaxSAT: hard and soft clauses

- **Hard clauses** – **must be satisfied**
- **Soft clauses** – we **would like to satisfy**, *if possible*
  - Associate a **cost** with *falsifying* each soft clause
  - For a hard clause, the cost can be viewed as ∞

Problem: find an assignment that satisfies all hard clauses and maximizes the sum of weights of satisfied soft clauses.

- **Alternatively, can be seen a typical optimisation problem:**

$$\min \quad \sum_{j=1}^{n} c_j x_j$$

$$\text{s.t.} \quad \varphi$$

feasibility region

- **Hard clauses φ**

General MaxSAT: hard and soft clauses

- **Hard clauses** – **must be satisfied**
- **Soft clauses** – we **would like to satisfy**, *if possible*
  - Associate a **cost** with *falsifying* each soft clause
  - For a hard clause, the cost can be viewed as ∞

Problem: find an assignment that satisfies all hard clauses and maximizes the sum of weights of satisfied soft clauses.

- **Alternatively, can be seen a typical optimisation problem:**

$$\min \quad \sum_{j=1}^{n} c_j x_j$$

$$\text{s.t.} \quad \varphi$$

feasibility region

objective function

- **Hard clauses φ**
- **Soft clauses $(\neg x_j)$ with cost c_j**

Hard and soft clauses — example

$$\begin{aligned} \max \quad & 2x - 3y + 5z \\ \text{s.t.} \quad & (\neg x \vee \neg y) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg z) \end{aligned}$$

Hard and soft clauses — example

$$\begin{aligned} \max \quad & 2x - 3y + 5z \\ \text{s.t.} \quad & (\neg x \vee \neg y) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg z) \end{aligned}$$



$$(\neg x \vee \neg y, \infty) \wedge (\neg x \vee \neg z, \infty) \wedge (\neg y \vee \neg z, \infty)$$

$$\wedge$$

$$(x, 2) \wedge (\neg y, 3) \wedge (z, 5)$$

Hard and soft clauses — DIMACS WCNF format

- `p wcnf N M T` – **preamble**
  - N – number of variables
  - M – number of clauses

Hard and soft clauses — DIMACS WCNF format

- `p wcnf N M T` – **preamble**
  - N – number of variables
  - M – number of clauses
  - T – ***top weight***, i.e. "∞"
  - T normally equals $1 + \sum_{c_i \text{ is soft}} w_i$

Hard and soft clauses — DIMACS WCNF format

- `p wcnf N M T` – **preamble**
  - N – number of variables
  - M – number of clauses
  - T – ***top weight***, i.e. "∞"
  - T normally equals $1 + \sum_{c_i \text{ is soft}} w_i$
- `W C 0` – **clause**
  - W – weight of the clause
  - C – list of clause literals

Hard and soft clauses — DIMACS WCNF format

- `p wcnf N M T` – **preamble**
  - N – number of variables
  - M – number of clauses
  - T – ***top weight***, i.e. "∞"
  - T normally equals $1 + \sum_{c_i \text{ is soft}} w_i$
- `W C 0` – **clause**
  - W – weight of the clause
  - C – list of clause literals

$$(\neg x \vee \neg y, \infty) \wedge (\neg x \vee \neg z, \infty) \wedge (\neg y \vee \neg z, \infty)$$
$$\wedge$$
$$(x, 2) \wedge (\neg y, 3) \wedge (z, 5)$$

```
1 p wcnf 3 6 11
2 11 -1 -2 0
3 11 -1 -3 0
4 11 -2 -3 0
5 2 1 0
6 3 -2 0
7 5 3 0
```

Hard and soft clauses — DIMACS WCNF format

New format gets rid of (i) preamble and (ii) top weight

- `p wcnf N M T` – **preamble**
  - N – number of variables
  - M – number of clauses
  - T – ***top weight***, i.e. "∞"
  - T normally equals $1 + \sum_{c_i \text{ is soft}} w_i$
- `W C 0` – **clause**
  - W – weight of the clause
  - C – list of clause literals

```
h -1 -2 0
h -1 -3 0
h -2 -3 0
2 1 0
3 -2 0
5 3 0
```

$$(\neg x \vee \neg y, \infty) \wedge (\neg x \vee \neg z, \infty) \wedge (\neg y \vee \neg z, \infty)$$
$$\wedge$$
$$(x, 2) \wedge (\neg y, 3) \wedge (z, 5)$$

```
p wcnf 3 6 11
11 -1 -2 0
11 -1 -3 0
11 -2 -3 0
2 1 0
3 -2 0
5 3 0
```

Why MaxSAT?

A powerful declarative paradigm for solving complex optimization problems across various domains!

Why MaxSAT?

A powerful declarative paradigm for solving complex optimization problems across various domains!

- Planning, scheduling, routing;
- Package dependency management;
- Knowledge compilation, representation, and reasoning;
- Model-based diagnosis and software fault localization;
- Axiom pinpointing and debugging;
- Graph optimisation problems;
- Bioinformatics and computational biology;
- Correlation clustering;
- Causal discovery;
- Interpretable ML model infererence;
- Formal eXplainable AI (FXAI);
- many, many more!..

Why MaxSAT?

A powerful declarative paradigm for solving complex optimization problems across various domains!

- Planning, scheduling, routing;
- Package dependency management;
- Knowledge compilation, representation, and reasoning;
- Model-based diagnosis and software fault localization;
- Axiom pinpointing and debugging;
- Graph optimisation problems;
- Bioinformatics and computational biology;
- Correlation clustering;
- Causal discovery;
- Interpretable ML model infererence;
- Formal eXplainable AI (FXAI);
- many, many more!..

The Need for Speed! :-)

The wide applicability drives the demand for highly efficient and robust solvers.

Example — solving MaxClique with MaxSAT

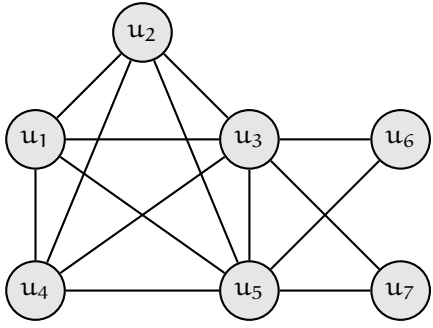
Construct $\mathsf{F} = \langle H, S \rangle$

Example — solving MaxClique with MaxSAT

$$\text{Construct } F = \langle H, S \rangle \text{ s.t. } \begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$$

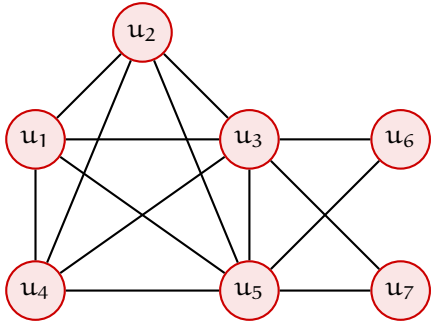
Example — solving MaxClique with MaxSAT

Construct $F = \langle H, S \rangle$ s.t. $\begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$



Example — solving MaxClique with MaxSAT

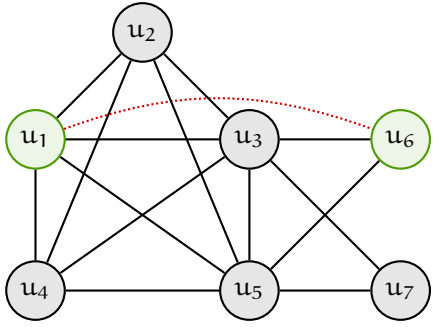
$$\text{Construct } F = \langle H, S \rangle \text{ s.t. } \begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$$



$$S = \left\{ \begin{array}{l} (x_1)\,(x_2)\,(x_3) \\ (x_4)\,(x_5)\,(x_6) \\ (x_7) \end{array} \right\}$$

Example — solving MaxClique with MaxSAT

Construct $F = \langle H, S \rangle$ s.t. $\begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$

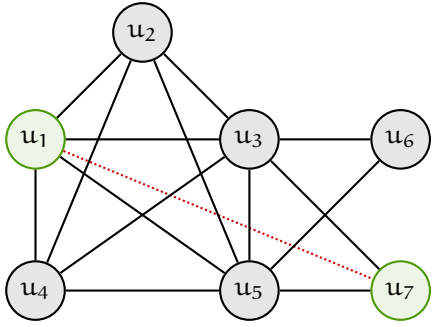


$$H = \left\{ \begin{array}{l} (\neg x_1 \vee \neg x_6)\ (\neg x_1 \vee \neg x_7) \\ (\neg x_2 \vee \neg x_6)\ (\neg x_2 \vee \neg x_7) \\ (\neg x_4 \vee \neg x_6)\ (\neg x_4 \vee \neg x_7) \\ (\neg x_6 \vee \neg x_7) \end{array} \right\}$$

$$S = \left\{ \begin{array}{l} (x_1)\ (x_2)\ (x_3) \\ (x_4)\ (x_5)\ (x_6) \\ (x_7) \end{array} \right\}$$

Example — solving MaxClique with MaxSAT

Construct $F = \langle H, S \rangle$ s.t. $\begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$

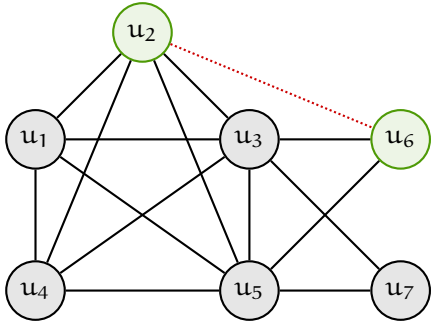


$$H = \left\{ \begin{array}{l} (\neg x_1 \vee \neg x_6)\ (\neg x_1 \vee \neg x_7) \\ (\neg x_2 \vee \neg x_6)\ (\neg x_2 \vee \neg x_7) \\ (\neg x_4 \vee \neg x_6)\ (\neg x_4 \vee \neg x_7) \\ (\neg x_6 \vee \neg x_7) \end{array} \right\}$$

$$S = \left\{ \begin{array}{l} (x_1)\ (x_2)\ (x_3) \\ (x_4)\ (x_5)\ (x_6) \\ (x_7) \end{array} \right\}$$

Example — solving MaxClique with MaxSAT

$$\text{Construct } F = \langle H, S \rangle \text{ s.t. } \begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$$

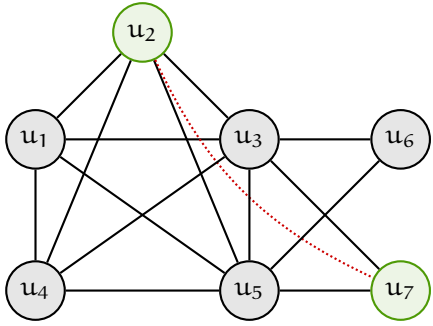


$$H = \left\{ \begin{array}{l} (\neg x_1 \vee \neg x_6)\,(\neg x_1 \vee \neg x_7) \\ (\neg x_2 \vee \neg x_6)\,(\neg x_2 \vee \neg x_7) \\ (\neg x_4 \vee \neg x_6)\,(\neg x_4 \vee \neg x_7) \\ (\neg x_6 \vee \neg x_7) \end{array} \right\}$$

$$S = \left\{ \begin{array}{l} (x_1)\,(x_2)\,(x_3) \\ (x_4)\,(x_5)\,(x_6) \\ (x_7) \end{array} \right\}$$

Example — solving MaxClique with MaxSAT

$$\text{Construct } F = \langle H, S \rangle \text{ s.t. } \begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$$

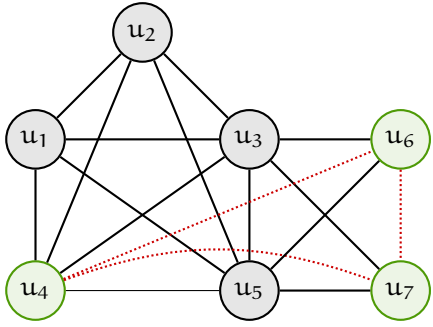


$$H = \left\{ \begin{array}{l} (\neg x_1 \vee \neg x_6)\,(\neg x_1 \vee \neg x_7) \\ (\neg x_2 \vee \neg x_6)\,(\neg x_2 \vee \neg x_7) \\ (\neg x_4 \vee \neg x_6)\,(\neg x_4 \vee \neg x_7) \\ (\neg x_6 \vee \neg x_7) \end{array} \right\}$$

$$S = \left\{ \begin{array}{l} (x_1)\,(x_2)\,(x_3) \\ (x_4)\,(x_5)\,(x_6) \\ (x_7) \end{array} \right\}$$

Example — solving MaxClique with MaxSAT

$$\text{Construct } F = \langle H, S \rangle \text{ s.t. } \begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$$

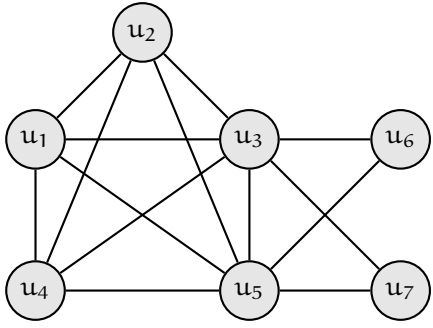


$$H = \left\{ \begin{array}{l} (\neg x_1 \vee \neg x_6)\,(\neg x_1 \vee \neg x_7) \\ (\neg x_2 \vee \neg x_6)\,(\neg x_2 \vee \neg x_7) \\ (\neg x_4 \vee \neg x_6)\,(\neg x_4 \vee \neg x_7) \\ (\neg x_6 \vee \neg x_7) \end{array} \right\}$$

$$S = \left\{ \begin{array}{l} (x_1)\,(x_2)\,(x_3) \\ (x_4)\,(x_5)\,(x_6) \\ (x_7) \end{array} \right\}$$

Example — solving MaxClique with MaxSAT

Construct $F = \langle H, S \rangle$ s.t. $\begin{cases} H & \triangleq & \{(\neg x_u \vee \neg x_v) \mid (u, v) \in E^C\} \\ S & \triangleq & \{(x_u) \mid v \in V\} \end{cases}$

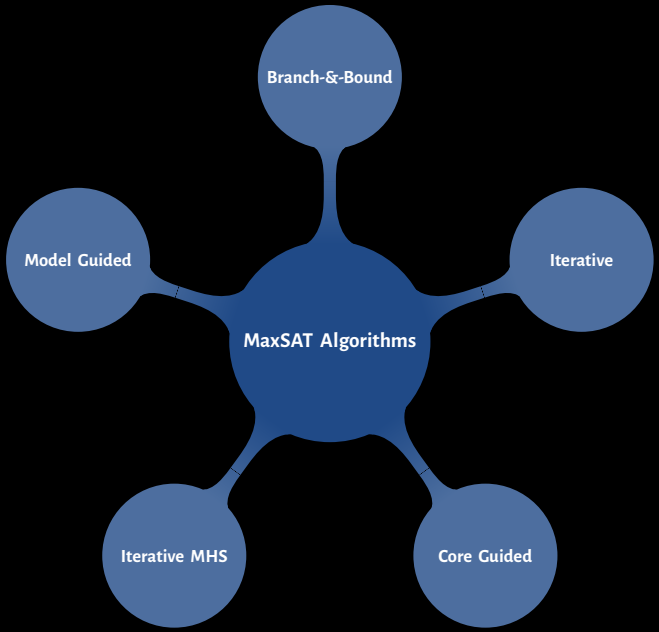


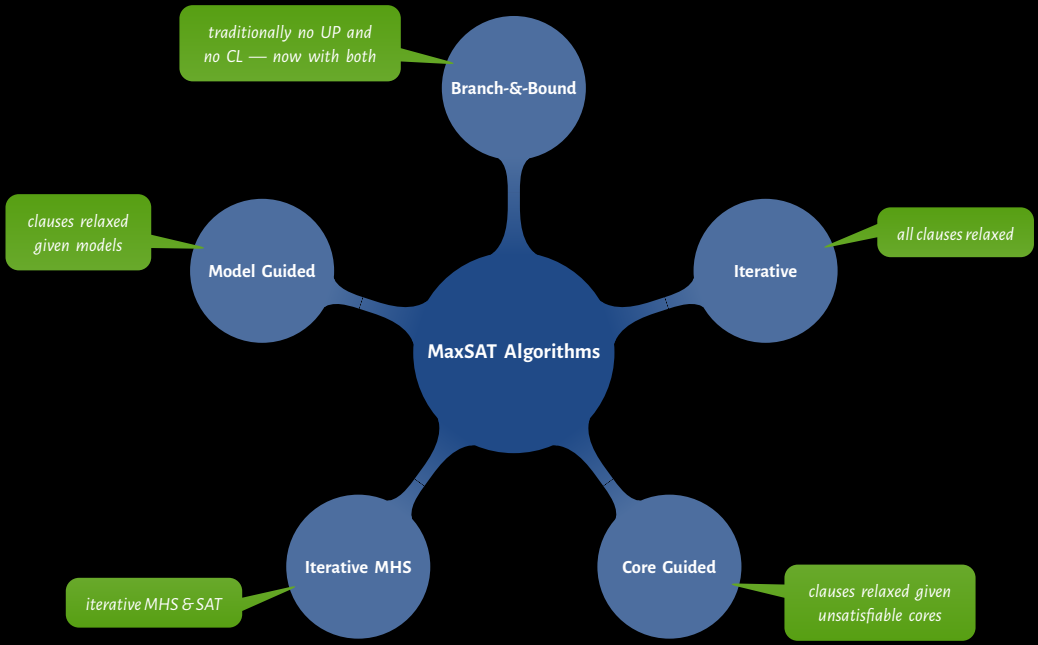
$$H = \begin{Bmatrix} (\neg x_1 \vee \neg x_6)\,(\neg x_1 \vee \neg x_7) \\ (\neg x_2 \vee \neg x_6)\,(\neg x_2 \vee \neg x_7) \\ (\neg x_4 \vee \neg x_6)\,(\neg x_4 \vee \neg x_7) \\ (\neg x_6 \vee \neg x_7) \end{Bmatrix}$$

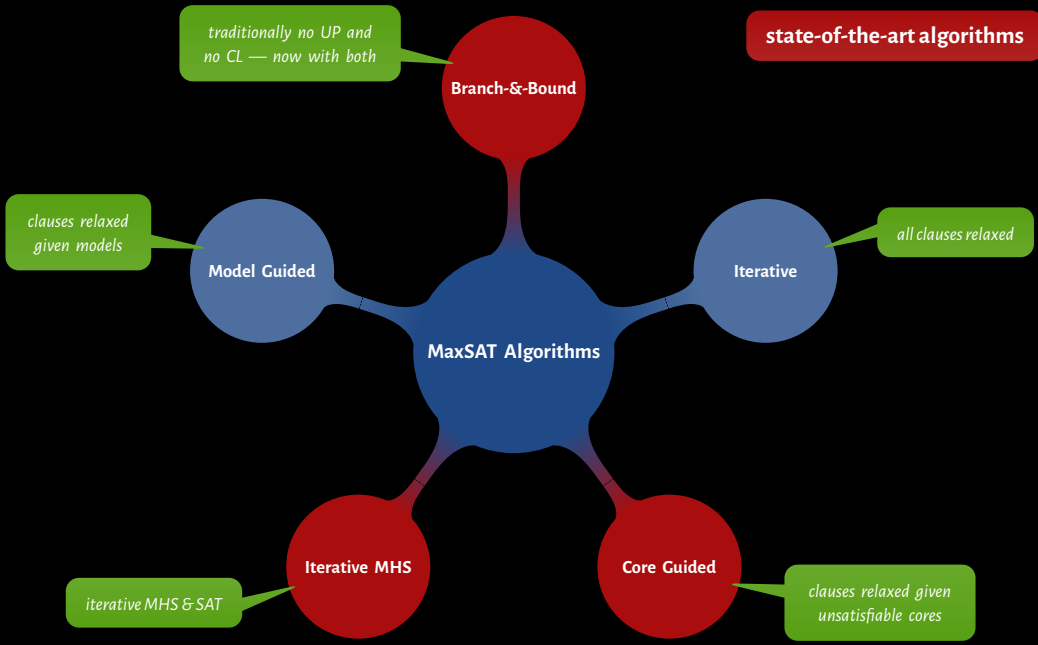
$$S = \begin{Bmatrix} (x_1)\,(x_2)\,(x_3) \\ (x_4)\,(x_5)\,(x_6) \\ (x_7) \end{Bmatrix}$$

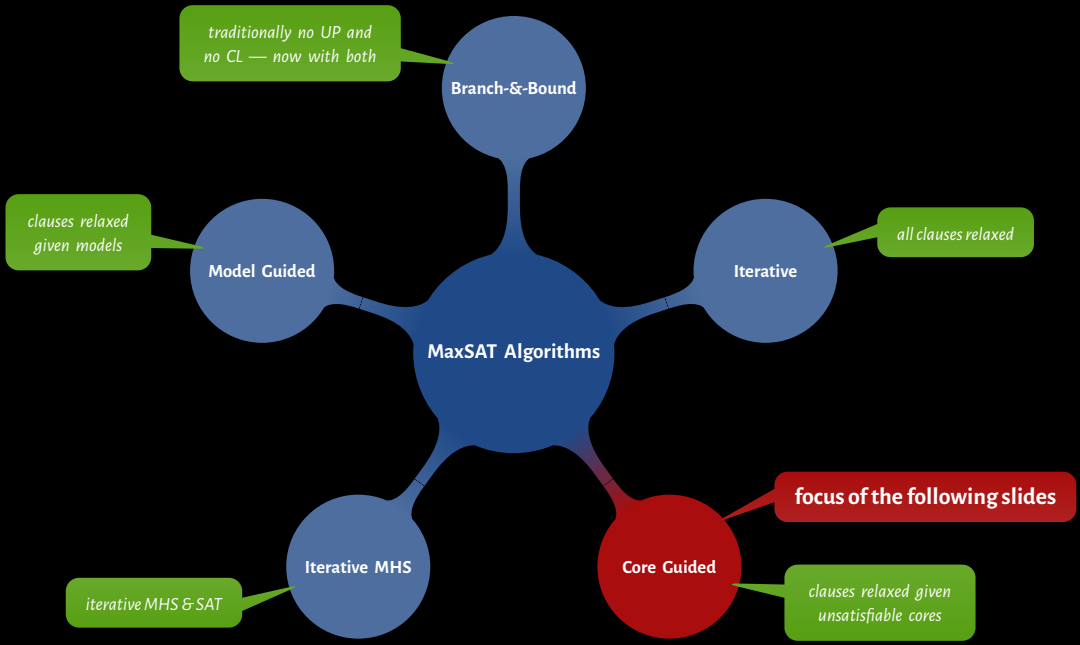
solve F with MaxSAT !

Solving MaxSAT


Branch-&-Bound
Model Guided
Iterative
MaxSAT Algorithms
Iterative MHS
Core Guided


MaxSAT Algorithms
Branch-&-Bound
traditionally no UP and no CL — now with both
Model Guided
clauses relaxed given models
Iterative
all clauses relaxed
Iterative MHS
iterative MHS & SAT
Core Guided
clauses relaxed given unsatisfiable cores


MaxSAT Algorithms
Model Guided
clauses relaxed given models
Iterative
all clauses relaxed
Branch-&-Bound
traditionally no UP and no CL — now with both
state-of-the-art algorithms
Iterative MHS
iterative MHS & SAT
Core Guided
clauses relaxed given unsatisfiable cores


MaxSAT Algorithms
Branch-&-Bound
traditionally no UP and no CL — now with both
Model Guided
clauses relaxed given models
Iterative
all clauses relaxed
Iterative MHS
iterative MHS & SAT
Core Guided
clauses relaxed given unsatisfiable cores
focus of the following slides

main idea of iterative MaxSAT:

relax all soft clauses:

$$\mathcal{S}' \leftarrow \{c_i \vee r_i \mid c_i \in \mathcal{S}\}$$

iterate through k:

$$\text{SAT}\left(\mathcal{H} \wedge \mathcal{S}' \wedge \sum_{i \in |\mathcal{S}'|} r_i \leq k\right)$$

main idea of iterative MaxSAT:

relax all soft clauses:

$$\mathcal{S}' \leftarrow \{c_i \vee r_i \mid c_i \in \mathcal{S}\}$$

iterate through k:

$$\text{SAT}\left(\mathcal{H} \wedge \mathcal{S}' \wedge \sum_{i \in |\mathcal{S}'|} r_i \leq k\right)$$

note that $|S'|$ and k may be too large!

this significantly affects solver's performance!

this is what core-guided algorithms help avoid

What is a core?

A set of clauses $\kappa \subseteq \mathcal{S}$ is an unsatisfiable core of $\phi = \mathcal{H} \wedge \mathcal{S}$ if $\mathcal{H} \wedge \kappa$ is unsatisfiable.

$$\mathcal{H} = \left\{ \begin{array}{llll} (x_6 \vee x_2 \vee r_1) & (\neg x_6 \vee x_2 \vee r_2) & (\neg x_2 \vee x_1 \vee r_3) & (\neg x_6 \vee x_8 \vee r_4) \quad (x_6 \vee \neg x_8 \vee r_5) \\ (x_2 \vee x_4 \vee r_6) & (\neg x_4 \vee x_5 \vee r_7) & (x_7 \vee x_5 \vee r_8) & (\neg x_7 \vee x_5 \vee r_9) \quad (\neg x_5 \vee x_3 \vee r_{10}) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{llll} (\neg r_1) & (\neg r_2) & (\neg r_3) & (\neg x_1) \\ (\neg r_4) & (\neg r_5) & (\neg r_6) & (\neg r_7) \\ (\neg r_8) & (\neg r_9) & (\neg r_{10}) & (\neg x_3) \end{array} \right\}$$

What is a core?

A set of clauses $\kappa \subseteq \mathcal{S}$ is an unsatisfiable core of $\phi = \mathcal{H} \wedge \mathcal{S}$ if $\mathcal{H} \wedge \kappa$ is unsatisfiable.

$$\mathcal{H} = \left\{ \begin{array}{llllll} (x_6 \vee x_2 \vee r_1) & (\neg x_6 \vee x_2 \vee r_2) & (\neg x_2 \vee x_1 \vee r_3) & (\neg x_6 \vee x_8 \vee r_4) & (x_6 \vee \neg x_8 \vee r_5) \\ (x_2 \vee x_4 \vee r_6) & (\neg x_4 \vee x_5 \vee r_7) & (x_7 \vee x_5 \vee r_8) & (\neg x_7 \vee x_5 \vee r_9) & (\neg x_5 \vee x_3 \vee r_{10}) \end{array} \right\}$$

κ_1

$$\mathcal{S} = \left\{ \begin{array}{llll} (\neg r_1) & (\neg r_2) & (\neg r_3) & (\neg x_1) \\ (\neg r_4) & (\neg r_5) & (\neg r_6) & (\neg r_7) \\ (\neg r_8) & (\neg r_9) & (\neg r_{10}) & (\neg x_3) \end{array} \right\}$$

What is a core?

A set of clauses $\kappa \subseteq \mathcal{S}$ is an unsatisfiable core of $\phi = \mathcal{H} \wedge \mathcal{S}$ if $\mathcal{H} \wedge \kappa$ is unsatisfiable.

$$\mathcal{H} = \left\{ \begin{array}{llllll} (x_6 \vee x_2 \vee r_1) & (\neg x_6 \vee x_2 \vee r_2) & (\neg x_2 \vee x_1 \vee r_3) & (\neg x_6 \vee x_8 \vee r_4) & (x_6 \vee \neg x_8 \vee r_5) \\ (x_2 \vee x_4 \vee r_6) & (\neg x_4 \vee x_5 \vee r_7) & (x_7 \vee x_5 \vee r_8) & (\neg x_7 \vee x_5 \vee r_9) & (\neg x_5 \vee x_3 \vee r_{10}) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{llll} (\neg r_1) & (\neg r_2) & (\neg r_3) & (\neg x_1) \\ (\neg r_4) & (\neg r_5) & (\neg r_6) & (\neg r_7) \\ (\neg r_8) & (\neg r_9) & (\neg r_{10}) & (\neg x_3) \end{array} \right\}$$

κ_2

What is a core?

A set of clauses $\kappa \subseteq \mathcal{S}$ is an unsatisfiable core of $\phi = \mathcal{H} \wedge \mathcal{S}$ if $\mathcal{H} \wedge \kappa$ is unsatisfiable.

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_6 \vee x_2 \vee r_1) & (\neg x_6 \vee x_2 \vee r_2) & (\neg x_2 \vee x_1 \vee r_3) & (\neg x_6 \vee x_8 \vee r_4) \quad (x_6 \vee \neg x_8 \vee r_5) \\ (x_2 \vee x_4 \vee r_6) & (\neg x_4 \vee x_5 \vee r_7) & (x_7 \vee x_5 \vee r_8) & (\neg x_7 \vee x_5 \vee r_9) \quad (\neg x_5 \vee x_3 \vee r_{10}) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{cccc} (\neg r_1) & (\neg r_2) & (\neg r_3) & (\neg x_1) \\ (\neg r_4) & (\neg r_5) & (\neg r_6) & (\neg r_7) \\ (\neg r_8) & (\neg r_9) & (\neg r_{10}) & (\neg x_3) \end{array} \right\}$$

κ_3

Z. Fu & S. Malik pioneered core-guided MaxSAT

it revolutionized MaxSAT solving

Z. Fu & S. Malik pioneered core-guided MaxSAT

it revolutionized MaxSAT solving

Z. Fu & S. Malik pioneered core-guided MaxSAT

many core-guided algorithms since then:

MSU1, PM1, PM1.2,
MSU2, MSU3, MSU4,
BIN-C, BIN-C-D, PRG,
WPM3,
OLL,
PMRES,
K / Maxino,
…

it revolutionized MaxSAT solving

Z. Fu & S. Malik pioneered core-guided MaxSAT

basic idea:

```
while not SAT(φ):
    κ ← get_core(φ)
    ρ ← relax(φ,κ)
    φ ← φ ∪ ∑_{r∈ρ} r ⩽ 1
return get_model(φ)
```

many core-guided algorithms since then:

MSU1, PM1, PM1.2,
MSU2, MSU3, MSU4,
BIN-C, BIN-C-D, PRG,
WPM3,
OLL,
PMRES,
K / Maxino,
…

it revolutionized MaxSAT solving

Z. Fu & S. Malik pioneered core-guided MaxSAT

identify and resolve inconsistencies

basic idea:

```
while not SAT(φ):
    κ ← get_core(φ)
    ρ ← relax(φ,κ)
    φ ← φ ∪ ∑_{r∈ρ} r ⩽ 1
return get_model(φ)
```

many core-guided algorithms since then:

MSU1, PM1, PM1.2,
MSU2, MSU3, MSU4,
BIN-C, BIN-C-D, PRG,
WPM3,
OLL,
PMRES,
K / Maxino,
…

Fu&Malik's (FM) algorithm

$x_6 \vee x_2$ $\quad \neg x_6 \vee x_2$ $\quad \neg x_2 \vee x_1$ $\quad \neg x_1$

$\neg x_6 \vee x_8$ $\quad x_6 \vee \neg x_8$ $\quad x_2 \vee x_4$ $\quad \neg x_4 \vee x_5$

$x_7 \vee x_5$ $\quad \neg x_7 \vee x_5$ $\quad \neg x_5 \vee x_3$ $\quad \neg x_3$

Example CNF formula ***(no hard clauses!)***

Fu&Malik's (FM) algorithm

$x_6 \vee x_2$ $\quad$ $\neg x_6 \vee x_2$ $\quad$ $\neg x_2 \vee x_1$ $\quad$ $\neg x_1$

$\neg x_6 \vee x_8$ $\quad$ $x_6 \vee \neg x_8$ $\quad$ $x_2 \vee x_4$ $\quad$ $\neg x_4 \vee x_5$

$x_7 \vee x_5$ $\quad$ $\neg x_7 \vee x_5$ $\quad$ $\neg x_5 \vee x_3$ $\quad$ $\neg x_3$

Formula is UNSAT; Get unsat core

Fu&Malik's (FM) algorithm

$$x_6 \vee x_2 \qquad \neg x_6 \vee x_2 \qquad \neg x_2 \vee x_1 \vee r_1 \qquad \neg x_1 \vee r_2$$

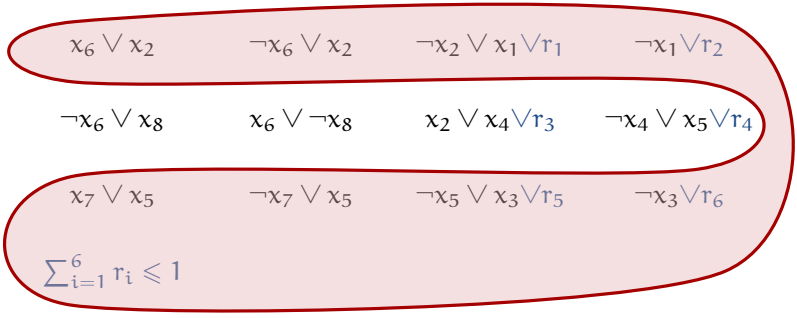
$$\neg x_6 \vee x_8 \qquad x_6 \vee \neg x_8 \qquad x_2 \vee x_4 \vee r_3 \qquad \neg x_4 \vee x_5 \vee r_4$$

$$x_7 \vee x_5 \qquad \neg x_7 \vee x_5 \qquad \neg x_5 \vee x_3 \vee r_5 \qquad \neg x_3 \vee r_6$$

$$\sum_{i=1}^{6} r_i \leqslant 1$$

Add relaxation variables and AtMost1 constraint

Fu&Malik's (FM) algorithm



Formula is (again) UNSAT; Get unsat core

Fu&Malik's (FM) algorithm

$$x_6 \vee x_2 \vee r_7 \qquad \neg x_6 \vee x_2 \vee r_8 \qquad \neg x_2 \vee x_1 \vee r_1 \vee r_9 \qquad \neg x_1 \vee r_2 \vee r_{10}$$

$$\neg x_6 \vee x_8 \qquad x_6 \vee \neg x_8 \qquad x_2 \vee x_4 \vee r_3 \qquad \neg x_4 \vee x_5 \vee r_4$$

$$x_7 \vee x_5 \vee r_{11} \qquad \neg x_7 \vee x_5 \vee r_{12} \qquad \neg x_5 \vee x_3 \vee r_5 \vee r_{13} \qquad \neg x_3 \vee r_6 \vee r_{14}$$

$$\sum_{i=1}^{6} r_i \leqslant 1 \qquad \sum_{i=7}^{14} r_i \leqslant 1$$

Add new relaxation variables and AtMost1 constraint

Fu&Malik's (FM) algorithm

$x_6 \vee x_2 \vee r_7$ $\quad \neg x_6 \vee x_2 \vee r_8$ $\quad \neg x_2 \vee x_1 \vee r_1 \vee r_9$ $\quad \neg x_1 \vee r_2 \vee r_{10}$

$\neg x_6 \vee x_8$ $\quad x_6 \vee \neg x_8$ $\quad x_2 \vee x_4 \vee r_3$ $\quad \neg x_4 \vee x_5 \vee r_4$

$x_7 \vee x_5 \vee r_{11}$ $\quad \neg x_7 \vee x_5 \vee r_{12}$ $\quad \neg x_5 \vee x_3 \vee r_5 \vee r_{13}$ $\quad \neg x_3 \vee r_6 \vee r_{14}$

$\sum_{i=1}^{6} r_i \leqslant 1$ $\quad \sum_{i=7}^{14} r_i \leqslant 1$

Formula is now SAT

Fu&Malik's (FM) algorithm

$x_6 \vee x_2 \vee r_7$ $\quad \neg x_6 \vee x_2 \vee r_8$ $\quad \neg x_2 \vee x_1 \vee r_1 \vee r_9$ $\quad \neg x_1 \vee r_2 \vee r_{10}$

$\neg x_6 \vee x_8$ $\quad x_6 \vee \neg x_8$ $\quad x_2 \vee x_4 \vee r_3$ $\quad \neg x_4 \vee x_5 \vee r_4$

$x_7 \vee x_5 \vee r_{11}$ $\quad \neg x_7 \vee x_5 \vee r_{12}$ $\quad \neg x_5 \vee x_3 \vee r_5 \vee r_{13}$ $\quad \neg x_3 \vee r_6 \vee r_{14}$

$\sum_{i=1}^{6} r_i \leqslant 1$ $\quad \sum_{i=7}^{14} r_i \leqslant 1$

MaxSAT cost is 2

Fu&Malik's (FM) algorithm

$x_6 \vee x_2 \vee r_7$ $\quad \neg x_6 \vee x_2 \vee r_8$ $\quad \neg x_2 \vee x_1 \vee r_1 \vee r_9$ $\quad \neg x_1 \vee r_2 \vee r_{10}$

$\neg x_6 \vee x_8$ $\quad x_6 \vee \neg x_8$ $\quad x_2 \vee x_4 \vee r_3$ $\quad \neg x_4 \vee x_5 \vee r_4$

$x_7 \vee x_5 \vee r_{11}$ $\quad \neg x_7 \vee x_5 \vee r_{12}$ $\quad \neg x_5 \vee x_3 \vee r_5 \vee r_{13}$ $\quad \neg x_3 \vee r_6 \vee r_{14}$

$\sum_{i=1}^{6} r_i \leqslant 1$ $\quad \sum_{i=7}^{14} r_i \leqslant 1$

MaxSAT cost is 2

Only AtMost1 constraints used

Relaxed soft clauses remain *soft*

Fu&Malik's (FM) algorithm

$x_6 \vee x_2 \vee r_7$ $\quad \neg x_6 \vee x_2 \vee r_8$ $\quad \neg x_2 \vee x_1 \vee r_1 \vee r_9$ $\quad \neg x_1 \vee r_2 \vee r_{10}$

$\neg x_6 \vee x_8$ $\quad x_6 \vee \neg x_8$ $\quad x_2 \vee x_4 \vee r_3$ $\quad \neg x_4 \vee x_5 \vee r_4$

$x_7 \vee x_5 \vee r_{11}$ $\quad \neg x_7 \vee x_5 \vee r_{12}$ $\quad \neg x_5 \vee x_3 \vee r_5 \vee r_{13}$ $\quad \neg x_3 \vee r_6 \vee r_{14}$

$\sum_{i=1}^{6} r_i \leqslant 1$ $\quad \sum_{i=7}^{14} r_i \leqslant 1$

MaxSAT cost is 2

Only AtMost1 constraints used

Some clauses *not relaxed*

Relaxed soft clauses remain *soft*

Relaxable Cardinality Constraints

the key novelty of the OLL algorithm is that

the key novelty of the OLL algorithm is that

it relaxes cardinality constraints

the idea was originally proposed for ASP

the key novelty of the OLL algorithm is that

it relaxes cardinality constraints

the idea was originally proposed for ASP

the key novelty of the OLL algorithm is that

it relaxes cardinality constraints

i.e. each cardinality constraint $\sum_{\kappa} r \leq k$

- is kept soft,
- can participate in future cores,
- and *serve as input* to other cardinality constraints

the idea was originally proposed for ASP

the key novelty of the OLL algorithm is that

it relaxes cardinality constraints

i.e. each cardinality constraint $\sum_{\kappa} r \leq k$

- is kept soft,
- can participate in future cores,
- and *serve as input* to other cardinality constraints

best performing core-guided algorithm

Example: Relaxable Cardinality Constraints

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{ccc} (\neg x_1) & (\neg x_2) & (\neg x_3) \\ (\neg x_4) & (\neg x_5) & \end{array} \right\}$$

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{ccc} (\neg x_1) & (\neg x_2) & (\neg x_3) \\ (\neg x_4) & (\neg x_5) & \end{array} \right\}$$

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{ccc} (\neg x_1) & (\neg x_2) & (\neg x_3) \\ (\neg x_4) & (\neg x_5) & \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable;

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{ccc} (\neg x_1) & (\neg x_2) & (\neg x_3) \\ (\neg x_4) & (\neg x_5) & \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{c} (x_1 + x_2 + x_3 + x_4 \leqslant 1) \\ (\neg x_5) \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2:

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{c} (x_1 + x_2 + x_3 + x_4 \leqslant 1) \\ (\neg x_5) \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{c} (x_1 + x_2 + x_3 + x_4 \leqslant 2) \\ (\neg x_5) \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$
iteration 3:

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{llll} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{c} (x_1 + x_2 + x_3 + x_4 \leqslant 2) \\ (\neg x_5) \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$
iteration 3: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_3 = \{x_1 + x_2 + x_3 + x_4 \leqslant 2, \neg x_5\}$

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{c} (x_1 + x_2 + x_3 + x_4 \leqslant 3) \\ (x_5 + \neg\,(x_1 + x_2 + x_3 + x_4 \leqslant 2) \leqslant 1) \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$
iteration 3: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_3 = \{x_1 + x_2 + x_3 + x_4 \leqslant 2, \neg x_5\}$
iteration 4:

Example: Relaxable Cardinality Constraints

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{c} (x_1 + x_2 + x_3 + x_4 \leqslant 3) \\ (x_5 + \neg\,(x_1 + x_2 + x_3 + x_4 \leqslant 2) \leqslant 1) \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$
iteration 3: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_3 = \{x_1 + x_2 + x_3 + x_4 \leqslant 2, \neg x_5\}$
iteration 4: $\mathcal{H} \wedge \mathcal{S}$ is satisfiable; model $\mu = \{x_1, \neg x_2, x_3, \neg x_4, x_5\}$

Core Processing as Objective Reformulation

given a core $\kappa \subseteq \mathcal{S}$, *counting* variables $\Upsilon \triangleq \{y_1, y_2, \ldots, y_{|\kappa|}\}$ are introduced:

Core Processing as Objective Reformulation

given a core $\kappa \subseteq \mathcal{S}$, *counting* variables $\Upsilon \triangleq \{y_1, y_2, \ldots, y_{|\kappa|}\}$ are introduced:

$$\left(\sum_{x \in \kappa} x \geq j\right) \rightarrow y_j$$

Core Processing as Objective Reformulation

given a core $\kappa \subseteq \mathcal{S}$, *counting* variables $\Upsilon \triangleq \{y_1, y_2, \ldots, y_{|\kappa|}\}$ are introduced:

$$\left(\sum_{x \in \kappa} x \geq j\right) \rightarrow y_j$$

usually done with *iterative totalizer*

Core Processing as Objective Reformulation

given a core $\kappa \subseteq \mathcal{S}$, *counting* variables $\Upsilon \triangleq \{y_1, y_2, \ldots, y_{|\kappa|}\}$ are introduced:

usually done with *iterative totalizer*

$$\left(\sum_{x \in \kappa} x \geq j\right) \rightarrow y_j$$

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=3)
>>> print(t.rhs)
[6, 7, 8]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]
[-3, -5, 8]
```

Core Processing as Objective Reformulation

given a core $\kappa \subseteq \mathcal{S}$, *counting* variables $\Upsilon \triangleq \{y_1, y_2, \ldots, y_{|\kappa|}\}$ are introduced:

usually done with *iterative totalizer*

$$\left(\sum_{x \in \kappa} x \geq j\right) \rightarrow y_j$$

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=3)
>>> print(t.rhs)
[6, 7, 8]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]
[-3, -5, 8]
```

$$(x_1 + x_2 + x_3 \geqslant 1) \rightarrow x_6$$
$$(x_1 + x_2 + x_3 \geqslant 2) \rightarrow x_7$$
$$(x_1 + x_2 + x_3 \geqslant 3) \rightarrow x_8$$

Core Processing as Objective Reformulation

given a core $\kappa \subseteq \mathcal{S}$, *counting* variables $\Upsilon \triangleq \{y_1, y_2, \ldots, y_{|\kappa|}\}$ are introduced:

usually done with *iterative totalizer*

$$\left(\sum_{x \in \kappa} x \geq j\right) \rightarrow y_j$$

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=3)
>>> print(t.rhs)
[6, 7, 8]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]
[-3, -5, 8]
```

$$(x_1 + x_2 + x_3 \geqslant 1) \rightarrow x_6$$
$$(x_1 + x_2 + x_3 \geqslant 2) \rightarrow x_7$$
$$(x_1 + x_2 + x_3 \geqslant 3) \rightarrow x_8$$

$$\sum_{x \in \kappa} x = \sum_{j \in \{1, \ldots, |\kappa|\}} y_j$$

Core Processing as Objective Reformulation

given a core $\kappa \subseteq \mathcal{S}$, *counting* variables $\Upsilon \triangleq \{y_1, y_2, \ldots, y_{|\kappa|}\}$ are introduced:

usually done with *iterative totalizer*

$$\left(\sum_{x \in \kappa} x \geq j\right) \rightarrow y_j$$

```
1  >>> from pysat.card import *
2  >>> t = ITotalizer(lits=[1, 2, 3], ubound=3)
3  >>> print(t.rhs)
4  [6, 7, 8]
5  >>> print(*t.cnf.clauses, sep='\n')
6  [-2, 4]
7  [-1, 4]
8  [-1, -2, 5]
9  [-4, 6]
10 [-5, 7]
11 [-3, 6]
12 [-3, -4, 7]
13 [-3, -5, 8]
```

$$(x_1 + x_2 + x_3 \geqslant 1) \rightarrow x_6$$
$$(x_1 + x_2 + x_3 \geqslant 2) \rightarrow x_7$$
$$(x_1 + x_2 + x_3 \geqslant 3) \rightarrow x_8$$

$$\sum_{x \in \kappa} x = \sum_{j \in \{1,\ldots,|\kappa|\}} y_j$$

we know that $\sum_{x \in \kappa} x \geqslant 1$, so $y_1 = 1$

Core Processing as Objective Reformulation

given a core $\kappa \subseteq \mathcal{S}$, *counting* variables $\Upsilon \triangleq \{y_1, y_2, \ldots, y_{|\kappa|}\}$ are introduced:

usually done with *iterative totalizer*

$$\left(\sum_{x \in \kappa} x \geq j\right) \rightarrow y_j$$

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=3)
>>> print(t.rhs)
[6, 7, 8]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]
[-3, -5, 8]
```

$$(x_1 + x_2 + x_3 \geqslant 1) \rightarrow x_6$$
$$(x_1 + x_2 + x_3 \geqslant 2) \rightarrow x_7$$
$$(x_1 + x_2 + x_3 \geqslant 3) \rightarrow x_8$$

$$\sum_{x \in \kappa} x = \sum_{j \in \{1,\ldots,|\kappa|\}} y_j$$

we know that $\sum_{x \in \kappa} x \geqslant 1$, so $y_1 = 1$

i.e. $\sum_{x \in \kappa} x = 1 + \sum_{j \in \{2,\ldots,|\kappa|\}} y_j$

Example: Objective Reformulation

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{ccc} (\neg x_1) & (\neg x_2) & (\neg x_3) \\ (\neg x_4) & (\neg x_5) & \end{array} \right\}$$

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{ccc} (\neg x_1) & (\neg x_2) & (\neg x_3) \\ (\neg x_4) & (\neg x_5) & \end{array} \right\}$$

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{ccc} (\neg x_1) & (\neg x_2) & (\neg x_3) \\ (\neg x_4) & (\neg x_5) & \end{array} \right\}$$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable;

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{cccc} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

$$\mathcal{S} = \left\{ \begin{array}{ccc} (\neg x_1) & (\neg x_2) & (\neg x_3) \\ (\neg x_4) & (\neg x_5) & \end{array} \right\}$$

$\sum_{i=1}^{4} x_i = 1 + y_2 + y_3 + y_4$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{llll} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

represented as $(\neg y_2)$

$$\mathcal{S} = \left\{ \begin{array}{l} (x_1 + x_2 + x_3 + x_4 \leqslant 1) \\ (\neg x_5) \end{array} \right\}$$

minimise $y_2 + y_3 + y_4 + x_5 + 1$
subject to $\mathcal{H} \wedge \left[\bigwedge_{j=1}^{4} \left(\sum_{i=1}^{4} x_i \geqslant j\right) \rightarrow y_j\right]$

$\sum_{i=1}^{4} x_i = 1 + y_2 + y_3 + y_4$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2:

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{llll} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

represented as $(\neg y_2)$

$$\mathcal{S} = \left\{ \begin{array}{l} (x_1 + x_2 + x_3 + x_4 \leqslant 1) \\ (\neg x_5) \end{array} \right\}$$

minimise $y_2 + y_3 + y_4 + x_5 + 1$
subject to $\mathcal{H} \wedge \left[\bigwedge_{j=1}^{4} \left(\sum_{i=1}^{4} x_i \geqslant j\right) \rightarrow y_j\right]$

$\sum_{i=1}^{4} x_i = 1 + 1 + y_3 + y_4$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{llll} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

represented as $(\neg y_3)$

$$\mathcal{S} = \left\{ \begin{array}{l} (x_1 + x_2 + x_3 + x_4 \leqslant 2) \\ (\neg x_5) \end{array} \right\}$$

minimise $y_3 + y_4 + x_5 + 2$
subject to $\mathcal{H} \wedge \left[\bigwedge_{j=1}^{4} \left(\sum_{i=1}^{4} x_i \geqslant j \right) \rightarrow y_j \right]$

$\sum_{i=1}^{4} x_i = 1 + 1 + y_3 + y_4$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$
iteration 3:

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{llll} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

represented as $(\neg y_3)$

$$\mathcal{S} = \left\{ \begin{array}{l} (x_1 + x_2 + x_3 + x_4 \leqslant 2) \\ (\neg x_5) \end{array} \right\}$$

minimise $y_3 + y_4 + x_5 + 2$
subject to $\mathcal{H} \wedge \left[\bigwedge_{j=1}^{4} \left(\sum_{i=1}^{4} x_i \geqslant j \right) \rightarrow y_j \right]$

$\sum_{i=1}^{4} x_i = 1 + 1 + y_3 + y_4$

$y_3 + x_5 = 1 + z_2$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$
iteration 3: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_3 = \{x_1 + x_2 + x_3 + x_4 \leqslant 2, \neg x_5\}$

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{llll} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

represented as $(\neg y_4)$

$$\mathcal{S} = \left\{ \begin{array}{c} (x_1 + x_2 + x_3 + x_4 \leqslant 3) \\ (x_5 + \neg\,(x_1 + x_2 + x_3 + x_4 \leqslant 2) \leqslant 1) \end{array} \right\}$$

minimise $y_4 + z_2 + 3$
subject to $\mathcal{H} \wedge \left[\bigwedge_{j=1}^{4} \left(\sum_{i=1}^{4} x_i \geqslant j\right) \rightarrow y_j\right] \wedge \left[\bigwedge_{j=1}^{2} (y_3 + x_5 \geqslant j) \rightarrow z_j\right]$

represented as $(\neg z_2)$

$\sum_{i=1}^{4} x_i = 1 + 1 + y_3 + y_4$

$y_3 + x_5 = 1 + z_2$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$
iteration 3: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_3 = \{x_1 + x_2 + x_3 + x_4 \leqslant 2, \neg x_5\}$
iteration 4:

Example: Objective Reformulation

i.e. *minimise* $x_1 + x_2 + x_3 + x_4 + x_5$
subject to $\mathcal{H}$

$$\mathcal{H} = \left\{ \begin{array}{llll} (x_1 \vee x_2 \vee x_3 \vee x_4) & (x_1 \vee x_2) & (x_3 \vee x_4) & \\ (x_1 \vee x_5) & (x_2 \vee x_5) & (x_3 \vee x_5) & (x_4 \vee x_5) \end{array} \right\}$$

represented as $(\neg y_4)$

$$\mathcal{S} = \left\{ \begin{array}{l} (x_1 + x_2 + x_3 + x_4 \leqslant 3) \\ (x_5 + \neg\,(x_1 + x_2 + x_3 + x_4 \leqslant 2) \leqslant 1) \end{array} \right\}$$

minimise $y_4 + z_2 + 3$
subject to $\mathcal{H} \wedge \left[\bigwedge_{j=1}^{4} \left(\sum_{i=1}^{4} x_i \geqslant j\right) \rightarrow y_j\right] \wedge \left[\bigwedge_{j=1}^{2} (y_3 + x_5 \geqslant j) \rightarrow z_j\right]$

represented as $(\neg z_2)$

$\sum_{i=1}^{4} x_i = 1 + 1 + y_3 + y_4$

$y_3 + x_5 = 1 + z_2$

iteration 1: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_1 = \{\neg x_1, \neg x_2, \neg x_3, \neg x_4\}$
iteration 2: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_2 = \{x_1 + x_2 + x_3 + x_4 \leqslant 1\}$
iteration 3: $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable; core $\kappa_3 = \{x_1 + x_2 + x_3 + x_4 \leqslant 2, \neg x_5\}$
iteration 4: $\mathcal{H} \wedge \mathcal{S}$ is satisfiable; model $\mu = \{x_1, \neg x_2, x_3, \neg x_4, x_5\}$

weighted objective is handled similarly to FM

weighted objective is handled similarly to FM

by clause splitting / cloning

Some Practical Considerations

RC2 is a model open-source implementation of OLL

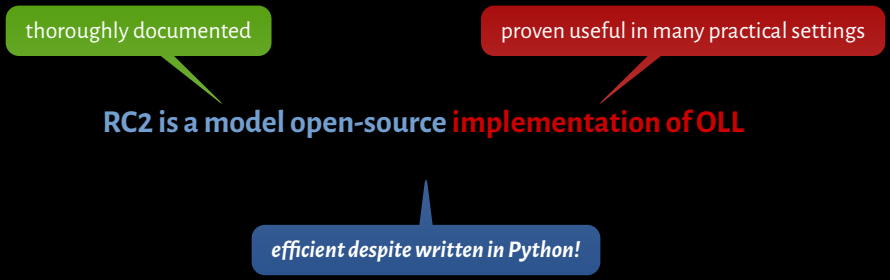
thoroughly documented

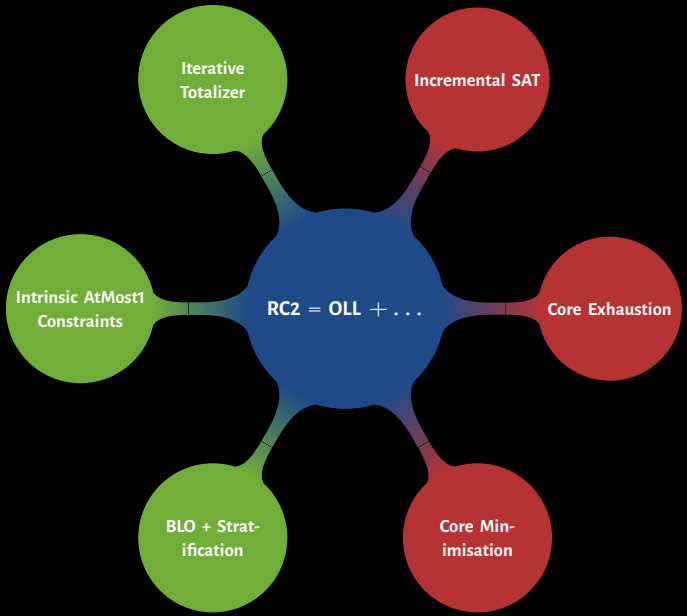
RC2 is a model open-source implementation of OLL

thoroughly documented

proven useful in many practical settings

RC2 is a model open-source implementation of OLL


thoroughly documented
proven useful in many practical settings
RC2 is a model open-source implementation of OLL
efficient despite written in Python!


Iterative Totalizer
Incremental SAT
Intrinsic AtMost1 Constraints
RC2 = OLL + . . .
Core Exhaustion
BLO + Strat-ification
Core Min-imisation

SAT solver is used incrementally

Incremental SAT

created once and never reinitialised

SAT solver is used incrementally

Incremental SAT

each non-unit $(c, w) \in \mathcal{S}$ is *augmented* with a selector:

created once and never reinitialised

SAT solver is used incrementally

Incremental SAT

each non-unit $(c, w) \in \mathcal{S}$ is *augmented* with a selector:
· s is a fresh variable;

created once and never reinitialised

SAT solver is used incrementally

Incremental SAT

each non-unit $(c, w) \in \mathcal{S}$ is *augmented* with a selector:
- s is a fresh variable;
- $c' \leftarrow (c \vee \neg s, \infty)$ added to $\mathcal{H}$

created once and never reinitialised

SAT solver is used incrementally

Incremental SAT

created once and never reinitialised

each non-unit $(c, w) \in \mathcal{S}$ is *augmented* with a selector:

- s is a fresh variable;
- $c' \leftarrow (c \vee \neg s, \infty)$ added to $\mathcal{H}$
- new (s, w) added to $\mathcal{S}$

SAT solver is used incrementally

Incremental SAT

each **non-unit** $(c, w) \in \mathcal{S}$ **is *augmented* with a selector:**

- s is a fresh variable;
- $c' \leftarrow (c \vee \neg s, \infty)$ added to $\mathcal{H}$
- new (s, w) added to $\mathcal{S}$

created once and never reinitialised

SAT solver is used incrementally

- SAT oracle is fed with hard clauses $\mathcal{H}$

Incremental SAT

each non-unit $(c, w) \in \mathcal{S}$ is *augmented* with a selector:

- s is a fresh variable;
- $c' \leftarrow (c \vee \neg s, \infty)$ added to $\mathcal{H}$
- new (s, w) added to $\mathcal{S}$

created once and never reinitialised

SAT solver is used incrementally

- SAT oracle is fed with hard clauses $\mathcal{H}$
- all $\ell \in \mathcal{S}$ are used as assumptions

Incremental SAT

each non-unit $(c, w) \in \mathcal{S}$ is *augmented* with a selector:

- s is a fresh variable;
- $c' \leftarrow (c \vee \neg s, \infty)$ added to $\mathcal{H}$
- new (s, w) added to $\mathcal{S}$

created once and never reinitialised

SAT solver is used incrementally

- SAT oracle is fed with hard clauses $\mathcal{H}$
- all $\ell \in \mathcal{S}$ are used as assumptions
- cardinality constraints $(\sum_{\ell \in \kappa} \geq j) \rightarrow y_j$ are treated as $\mathcal{H}$

Incremental SAT

each non-unit $(c, w) \in \mathcal{S}$ is *augmented* with a selector:
- s is a fresh variable;
- $c' \leftarrow (c \vee \neg s, \infty)$ added to $\mathcal{H}$
- new (s, w) added to $\mathcal{S}$

created once and never reinitialised

SAT solver is used incrementally

- SAT oracle is fed with hard clauses $\mathcal{H}$
- all $\ell \in \mathcal{S}$ are used as assumptions

- cardinality constraints $(\sum_{\ell \in \kappa} \geq j) \rightarrow y_j$ are treated as $\mathcal{H}$
- counting variables y_j are used as assumptions

Iterative Totalizer

given a core $\kappa \subseteq \mathcal{S}$, constraint $\sum_{\ell \in \kappa} \ell \leq j$ is encoded *incrementally*

Iterative Totalizer

given a core $\kappa \subseteq \mathcal{S}$, constraint $\sum_{\ell \in \kappa} \ell \leq j$ is encoded *incrementally*

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=1)
>>> print(t.rhs)
[6, 7]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]
```

Iterative Totalizer

given a core $\kappa \subseteq \mathcal{S}$, constraint $\sum_{\ell \in \kappa} \ell \leq j$ is encoded *incrementally*

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=1)
>>> print(t.rhs)
[6, 7]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]
```

$$(x_1 + x_2 + x_3 \geqslant 1) \rightarrow x_6$$
$$(x_1 + x_2 + x_3 \geqslant 2) \rightarrow x_7$$

Iterative Totalizer

given a core $\kappa \subseteq \mathcal{S}$, constraint $\sum_{\ell \in \kappa} \ell \leq j$ is encoded *incrementally*

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=1)
>>> print(t.rhs)
[6, 7]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]
```

assumption $\neg x_7$ is effectively used to enforce $(x_1 + x_2 + x_3 \leqslant 1)$

$$(x_1 + x_2 + x_3 \geqslant 1) \rightarrow x_6$$
$$(x_1 + x_2 + x_3 \geqslant 2) \rightarrow x_7$$

Iterative Totalizer

given a core $\kappa \subseteq \mathcal{S}$, constraint $\sum_{\ell \in \kappa} \ell \leq j$ is encoded *incrementally*

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=1)
>>> print(t.rhs)
[6, 7]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]

>>> t.increase(ubound=2)
>>> print(t.rhs)
[6, 7, 8]
>>> print(*t.cnf.clauses[-t.nof_new:], sep='\n')
[-3, -5, 8]
```

assumption $\neg x_7$ is effectively used to enforce $(x_1 + x_2 + x_3 \leqslant 1)$

$$(x_1 + x_2 + x_3 \geqslant 1) \rightarrow x_6$$
$$(x_1 + x_2 + x_3 \geqslant 2) \rightarrow x_7$$

Iterative Totalizer

given a core $\kappa \subseteq \mathcal{S}$, constraint $\sum_{\ell \in \kappa} \ell \leq j$ is encoded *incrementally*

```
>>> from pysat.card import *
>>> t = ITotalizer(lits=[1, 2, 3], ubound=1)
>>> print(t.rhs)
[6, 7]
>>> print(*t.cnf.clauses, sep='\n')
[-2, 4]
[-1, 4]
[-1, -2, 5]
[-4, 6]
[-5, 7]
[-3, 6]
[-3, -4, 7]

>>> t.increase(ubound=2)
>>> print(t.rhs)
[6, 7, 8]
>>> print(*t.cnf.clauses[-t.nof_new:], sep='\n')
[-3, -5, 8]
```

assumption $\neg x_7$ is effectively used to enforce $(x_1 + x_2 + x_3 \leqslant 1)$

$(x_1 + x_2 + x_3 \geqslant 1) \rightarrow x_6$

$(x_1 + x_2 + x_3 \geqslant 2) \rightarrow x_7$

now, the solver knows about $(x_1 + x_2 + x_3 \geqslant 3) \rightarrow x_8$

Core Exhaustion

unsatisfiable cores may form a (complex) tree structure

Core Exhaustion

unsatisfiable cores may form a (complex) tree structure

to simplify the tree structure, **RC2 tries to “exhaust” cores**

Core Exhaustion

unsatisfiable cores may form a (complex) tree structure

to simplify the tree structure, **RC2 tries to "exhaust" cores**

given a core $\kappa \subseteq \mathcal{S}$:

Core Exhaustion

unsatisfiable cores may form a (complex) tree structure

to simplify the tree structure, **RC2 tries to "exhaust" cores**

given a core $\kappa \subseteq \mathcal{S}$:

· find the largest k s.t. $\mathcal{H} \wedge \sum_{\ell \in \kappa} \ell \leq k$ is still UNSAT;

Core Exhaustion

unsatisfiable cores may form a (complex) tree structure

to simplify the tree structure, **RC2 tries to "exhaust" cores**

all other soft clauses are disregarded, i.e. the focus is solely on the core κ!

given a core $\kappa \subseteq \mathcal{S}$:

· find the largest k s.t. $\mathcal{H} \wedge \sum_{\ell \in \kappa} \ell \leq k$ is still UNSAT;

Core Exhaustion

unsatisfiable cores may form a (complex) tree structure

to simplify the tree structure, **RC2 tries to "exhaust" cores**

all other soft clauses are disregarded, i.e. the focus is solely on the core κ!

given a core $\kappa \subseteq \mathcal{S}$:

- find the largest k s.t. $\mathcal{H} \wedge \sum_{\ell \in \kappa} \ell \leq k$ is still UNSAT;
- if found, enforce constraint $\sum_{\ell \in \kappa} \ell \leq k + 1$;

Core Exhaustion

unsatisfiable cores may form a (complex) tree structure

to simplify the tree structure, **RC2 tries to "exhaust" cores**

all other soft clauses are disregarded, i.e. the focus is solely on the core κ!

given a core $\kappa \subseteq \mathcal{S}$:

- find the largest k s.t. $\mathcal{H} \wedge \sum_{\ell \in \kappa} \ell \leq k$ is still UNSAT;
- if found, enforce constraint $\sum_{\ell \in \kappa} \ell \leq k + 1$;

$k = 0$ if core exhaustion is disabled

Heuristic Core Minimisation

the smaller the core the better for solver's performance

traverse κ and try to remove literals

Heuristic Core Minimisation

the smaller the core the better for solver's performance

traverse κ and try to remove literals

```
def minimize_core(self):
    if len(self.core) > 1:
        self.core = sorted(self.core, key=lambda l: self.wght[l])
        self.oracle.conf_budget(1000)

        i = 0
        while i < len(self.core):
            to_test = self.core[:i] + self.core[(i + 1):]

            if self.oracle.solve_limited(assumptions=to_test) == False:
                self.core = to_test
            elif self.oracle.get_status() == True:
                i += 1
            else:
                break
```

Heuristic Core Minimisation

the smaller the core the better for solver's performance

traverse κ and try to remove literals

```
1  def minimize_core(self):
2      if len(self.core) > 1:
3          self.core = sorted(self.core, key=lambda l: self.wght[l])
4          self.oracle.conf_budget(1000)
5
6          i = 0
7          while i < len(self.core):
8              to_test = self.core[:i] + self.core[(i + 1):]
9
10             if self.oracle.solve_limited(assumptions=to_test) == False:
11                 self.core = to_test
12             elif self.oracle.get_status() == True:
13                 i += 1
14             else:
15                 break
```

is $\kappa \setminus \{l_i\}$ still a core?

Heuristic Core Minimisation

the smaller the core the better for solver's performance

traverse κ and try to remove literals

```
1  def minimize_core(self):
2      if len(self.core) > 1:
3          self.core = sorted(self.core, key=lambda l: self.wght[l])
4          self.oracle.conf_budget(1000)
5
6          i = 0
7          while i < len(self.core):
8              to_test = self.core[:i] + self.core[(i + 1):]
9
10             if self.oracle.solve_limited(assumptions=to_test) == False:
11                 self.core = to_test
12             elif self.oracle.get_status() == True:
13                 i += 1
14             else:
15                 break
```

overall budget is 1000 conflicts

is $\kappa \setminus \{l_i\}$ still a core?

Heuristic Core Minimisation

the smaller the core the better for solver's performance

traverse κ and try to remove literals

preference to remove literals with small weights

overall budget is 1000 conflicts

is $\kappa \setminus \{l_i\}$ still a core?

```
1  def minimize_core(self):
2      if len(self.core) > 1:
3          self.core = sorted(self.core, key=lambda l: self.wght[l])
4          self.oracle.conf_budget(1000)
5
6          i = 0
7          while i < len(self.core):
8              to_test = self.core[:i] + self.core[(i + 1):]
9
10             if self.oracle.solve_limited(assumptions=to_test) == False:
11                 self.core = to_test
12             elif self.oracle.get_status() == True:
13                 i += 1
14             else:
15                 break
```

weighted formulas are challenging

BLO and Stratification

clause splitting leads to a high number of iterations if weights are diverse

weighted formulas are challenging

BLO and Stratification

$\mathcal{S}_i = \{(c, w) \in \mathcal{S} \mid w = w_i\}, w_i > w_{i+1}$

clause splitting leads to a high number of iterations if weights are diverse

key idea:
- stratify the problem by weight
- solve it lexicographically, stratum by stratum

weighted formulas are challenging

BLO and Stratification

$\mathcal{S}_i = \{(c, w) \in \mathcal{S} \mid w = w_i\}, w_i > w_{i+1}$

key idea:
- stratify the problem by weight
- solve it lexicographically, stratum by stratum

clause splitting leads to a high number of iterations if weights are diverse

weighted formulas **are challenging**

iteratively solve:
- $\mathcal{H} \cup \mathcal{S}_1$

BLO and Stratification

$\mathcal{S}_i = \{(c, w) \in \mathcal{S} \mid w = w_i\}, w_i > w_{i+1}$

key idea:
- stratify the problem by weight
- solve it lexicographically, stratum by stratum

clause splitting leads to a high number of iterations if weights are diverse

weighted formulas **are challenging**

iteratively solve:
- $\mathcal{H} \cup \mathcal{S}_1$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2$

BLO and Stratification

$\mathcal{S}_i = \{(c, w) \in \mathcal{S} \mid w = w_i\}, w_i > w_{i+1}$

key idea:
- stratify the problem by weight
- solve it lexicographically, stratum by stratum

clause splitting leads to a high number of iterations if weights are diverse

weighted formulas **are challenging**

iteratively solve:
- $\mathcal{H} \cup \mathcal{S}_1$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2 \cup \ldots \cup \mathcal{S}_n$

BLO and Stratification

$\mathcal{S}_i = \{(c, w) \in \mathcal{S} \mid w = w_i\}, w_i > w_{i+1}$

key idea:
- stratify the problem by weight
- solve it lexicographically, stratum by stratum

clause splitting leads to a high number of iterations if weights are diverse

weighted formulas are challenging

this way we are *focusing* on the most important clauses first

reducing the impact of low-weighed clauses

iteratively solve:
- $\mathcal{H} \cup \mathcal{S}_1$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2 \cup \ldots \cup \mathcal{S}_n$

BLO and Stratification

$\mathcal{S}_i = \{(c, w) \in \mathcal{S} \mid w = w_i\}, w_i > w_{i+1}$

key idea:
- stratify the problem by weight
- solve it lexicographically, stratum by stratum

clause splitting **leads to a high number of iterations if weights are diverse**

weighted formulas are challenging

this way we are *focusing* on the most important clauses first

reducing the impact of low-weighed clauses

iteratively solve:
- $\mathcal{H} \cup \mathcal{S}_1$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2 \cup \ldots \cup \mathcal{S}_n$

BLO condition for stratum i:

$w_i > \sum_{j=i+1}^{n} w_j \times |\mathcal{S}_j|$

BLO and Stratification

$\mathcal{S}_i = \{(c, w) \in \mathcal{S} \mid w = w_i\}, w_i > w_{i+1}$

key idea:
- stratify the problem by weight
- solve it lexicographically, stratum by stratum

clause splitting **leads to a high number of iterations if weights are diverse**

weighted formulas are challenging

this way we are *focusing* on the most important clauses first

reducing the impact of low-weighed clauses

iteratively solve:
- $\mathcal{H} \cup \mathcal{S}_1$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2$
- $\mathcal{H} \cup \mathcal{S}_1 \cup \mathcal{S}_2 \cup \ldots \cup \mathcal{S}_n$

BLO condition for stratum i:

$w_i > \sum_{j=i+1}^{n} w_j \times |\mathcal{S}_j|$

clauses with weight $\geqslant w_i$ can be *hardened*

Intrinsic AtMost1 Constraints

$$\text{imagine } \exists \alpha \subseteq \mathcal{S} \text{ s.t. } \mathcal{H} \vDash \left(\sum_{\ell \in \alpha} \ell \leq 1\right)$$

Intrinsic AtMost1 Constraints

$|\alpha| \times (|\alpha| - 1)/2$ cores, each of size 2:
$\kappa = \{\ell_i, \ell_j\} \quad \forall \ell_i \neq \ell_j \in \alpha$

imagine $\exists \alpha \subseteq \mathcal{S}$ s.t. $\mathcal{H} \vDash \left(\sum_{\ell \in \alpha} \ell \leq 1\right)$

Intrinsic AtMost1 Constraints

if left "untreated", they form a complex tree structure

$|\alpha| \times (|\alpha| - 1)/2$ cores, each of size 2:
$\kappa = \{\ell_i, \ell_j\} \quad \forall \ell_i \neq \ell_j \in \alpha$

imagine $\exists \alpha \subseteq \mathcal{S}$ s.t. $\mathcal{H} \vDash \left(\sum_{\ell \in \alpha} \ell \leq 1\right)$

Intrinsic AtMost1 Constraints

can be *heuristically* detected using UP

if left "untreated", they form a complex tree structure

$|\alpha| \times (|\alpha| - 1)/2$ cores, each of size 2:
$\kappa = \{\ell_i, \ell_j\} \quad \forall \ell_i \neq \ell_j \in \alpha$

imagine $\exists \alpha \subseteq \mathcal{S}$ s.t. $\mathcal{H} \vDash \left(\sum_{\ell \in \alpha} \ell \leq 1\right)$

Intrinsic AtMost1 Constraints

```
1  from pysat.card import *
2  from pysat.solvers import *
3  a1 = CardEnc.atmost(lits=range(1, 11), bound=1)
4  >>> with Solver(bootstrap_with=a1) as solver:
5  ...     print(solver.propagate(assumptions=[1]))
6  ...
7  (True, [1, -2, -3, -4, -5, -6, -7, -8, -9, -10])
```

can be *heuristically* detected using UP

if left "untreated", they form a complex tree structure

$|\alpha| \times (|\alpha| - 1)/2$ cores, each of size 2: $\kappa = \{\ell_i, \ell_j\} \quad \forall \ell_i \neq \ell_j \in \alpha$

imagine $\exists \alpha \subseteq \mathcal{S}$ s.t. $\mathcal{H} \models \left(\sum_{\ell \in \alpha} \ell \leq 1\right)$

Intrinsic AtMost1 Constraints

contrast with $\kappa = \{x, y\}$ in
$(\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$

can be *heuristically* detected using UP

```
1  from pysat.card import *
2  from pysat.solvers import *
3  a1 = CardEnc.atmost(lits=range(1, 11), bound=1)
4  >>> with Solver(bootstrap_with=a1) as solver:
5  ...     print(solver.propagate(assumptions=[1]))
6  ...
7  (True, [1, -2, -3, -4, -5, -6, -7, -8, -9, -10])
```

if left "untreated", they form a complex tree structure

$|\alpha| \times (|\alpha| - 1)/2$ cores, each of size 2:
$\kappa = \{\ell_i, \ell_j\} \quad \forall \ell_i \neq \ell_j \in \alpha$

imagine $\exists \alpha \subseteq \mathcal{S}$ s.t. $\mathcal{H} \models \left(\sum_{\ell \in \alpha} \ell \leq 1\right)$

Intrinsic AtMost1 Constraints

contrast with $\kappa = \{x, y\}$ **in** $(\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$

can be *heuristically* detected using UP

```
1  from pysat.card import *
2  from pysat.solvers import *
3  a1 = CardEnc.atmost(lits=range(1, 11), bound=1)
4  >>> with Solver(bootstrap_with=a1) as solver:
5  ...     print(solver.propagate(assumptions=[1]))
6  ...
7  (True, [1, -2, -3, -4, -5, -6, -7, -8, -9, -10])
```

if left "untreated", they form a complex tree structure

$|\alpha| \times (|\alpha| - 1)/2$ cores, each of size 2: $\kappa = \{\ell_i, \ell_j\} \quad \forall \ell_i \neq \ell_j \in \alpha$

imagine $\exists \alpha \subseteq \mathcal{S}$ s.t. $\mathcal{H} \vDash \left(\sum_{\ell \in \alpha} \ell \leq 1\right)$

```
$ rc2.py -v frb20-11-1.partial.wcnf.gz
c formula: 220 vars, 5634 hard, 220 soft
c cost: 1; core sz: 2; soft sz: 219
c cost: 2; core sz: 2; soft sz: 218
...
c cost: 199; core sz: 5; soft sz: 9
c cost: 200; core sz: 6; soft sz: 8
s OPTIMUM FOUND
o 200
c oracle time: 160.3163
```

Intrinsic AtMost1 Constraints

contrast with $\kappa = \{x, y\}$ in
$(\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$

can be *heuristically* detected using UP

```
1 from pysat.card import *
2 from pysat.solvers import *
3 a1 = CardEnc.atmost(lits=range(1, 11), bound=1)
4 >>> with Solver(bootstrap_with=a1) as solver:
5 ...     print(solver.propagate(assumptions=[1]))
6 ...
7 (True, [1, -2, -3, -4, -5, -6, -7, -8, -9, -10])
```

if left "untreated", they form a complex tree structure

$|\alpha| \times (|\alpha| - 1)/2$ cores, each of size 2:
$\kappa = \{\ell_i, \ell_j\} \quad \forall \ell_i \neq \ell_j \in \alpha$

imagine $\exists \alpha \subseteq \mathcal{S}$ s.t. $\mathcal{H} \models \left(\sum_{\ell \in \alpha} \ell \leq 1\right)$

```
$ rc2.py -v frb20-11-1.partial.wcnf.gz
c formula: 220 vars, 5634 hard, 220 soft
c cost: 1; core sz: 2; soft sz: 219
c cost: 2; core sz: 2; soft sz: 218
...
c cost: 199; core sz: 5; soft sz: 9
c cost: 200; core sz: 6; soft sz: 8
s OPTIMUM FOUND
o 200
c oracle time: 160.3163
```

```
$ rc2.py -a -v frb20-11-1.partial.wcnf.gz
c formula: 220 vars, 5634 hard, 220 soft
c am1s found: 24; avgsz: 9.2; cost: 196
c cost: 197; core sz: 5; soft sz: 20
c cost: 198; core sz: 3; soft sz: 19
c cost: 199; core sz: 2; soft sz: 20
c cost: 200; core sz: 4; soft sz: 17
s OPTIMUM FOUND
o 200
c oracle time: 0.0021
```

exercise time

References i


R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum.
Diagnosing and solving over-determined constraint satisfaction problems.
In *IJCAI*, pages 276–281, 1993.


John W. Chinneck and Erik W. Dravnieks.
Locating minimal infeasible constraint sets in linear programs.
INFORMS Journal on Computing, 3(2):157–168, 1991.