

MiniSAT and SAT Encoding

CS402, Spring 2016
Shin Yoo

Boolean Satisfiability

- The problem of determining if there exists an interpretation that satisfies a given propositional formula.
- One of the first problems proven to be NP-complete.
- Often referred to simply as SAT

Computational Complexity

- 2-SAT refers to SAT problems, whose clauses only contain at most 2 literals. 2-SAT can be solved in polynomial time.
- 3-SAT refers to SAT problems whose clauses contain at most three literals.

$l_1 \vee \dots \vee l_n$ can be rewritten into 3-SAT as

$$(l_1 \vee l_2 \vee x_2) \wedge (\neg x_2 \vee l_3 \vee x_3) \wedge (\neg x_3 \vee l_4 \vee x_4) \wedge \dots \\ \wedge (\neg x_{n-3} \vee l_{n-2} \vee x_{n-2}) \wedge (\neg x_{n-2} \vee l_{n-1} \vee l_n)$$

Equisatisfiable

- Two formulas are equisatisfiable if the formula is satisfiable whenever the second is, and vice versa.
- This is different from logical equivalence, which states that two formulas have the same models; equisatisfiable formulas have different models.
- 3-SAT conversion is equisatisfiable.

MiniSAT

- Award winning open-source SAT solver
- <http://minisat.se>



The screenshot shows the homepage of 'The MiniSat Page'. The title 'The MiniSat Page' is in large blue letters at the top. Below it, a navigation menu on the left lists: Main, MiniSat, MiniSat+, SatELite, Papers, Authors, and Links. The main content area has a header 'INTRODUCTION' and a paragraph describing MiniSAT as a minimalistic, open-source SAT solver. To the right of this text is a photo of four trophies. Below the introduction is a section 'Some key features of MiniSAT:' with three bullet points: 'Easy to modify', 'Highly efficient', and 'Designed for integration'. Further down is a paragraph about community support and contact information. At the bottom is a 'NEWS' section with a list of updates from newest to oldest.

The MiniSat Page
by Niklas Eén, Niklas Sörensson

INTRODUCTION

MINISAT is a minimalistic, open-source SAT solver, developed to help researchers and developers alike to get started on SAT. It is released under the MIT licence, and is currently used in a number of projects (see "Links"). On this page you will find binaries, sources, documentation and projects related to MINISAT, including the Pseudo-boolean solver MiniSAT+ and the CNF minimizer/preprocessor SatELite. Together with SatELite, MiniSAT was recently awarded in the three [industrial](#) categories and one of the "crafted" categories of the SAT 2005 competition (see picture).



Some key features of MINISAT:

- **Easy to modify.** MINISAT is small and well-documented, and possibly also well-designed, making it an ideal starting point for adapting SAT based techniques to domain specific problems.
- **Highly efficient.** Winning all the industrial categories of the SAT 2005 competition, MINISAT is a good starting point both for future research in SAT, and for applications using SAT.
- **Designed for integration.** MINISAT supports incremental SAT and has mechanisms for adding non-clausal constraints. By virtue of being easy to modify, it is a good choice for integrating as a backend to another tool, such as a model checker or a more generic constraint solver.

We would like to start a community to help distribute knowledge about how to use and modify MINISAT. Any questions, comments, bugreports, bugfixes etc. should be directed to minisat@googlegroups.com. The archives can be browsed [here](#). The source code repository for MiniSat 2.2 can be found at [github](#).

— Niklas & Niklas

NEWS

From newest to oldest...

- A [paper](#) on how to compute localization abstractions using the incremental interface of MiniSat. View this as an example of a non-trivial incremental SAT application. A key feature of the algorithm is the use of `analyzeFinal()` to get the subset of assumptions used in the UNSAT proof.
- Finally! A new release of MiniSat, [downloads/minisat-2.2.0.tar.gz](#). For more info, see the [release notes](#).
- Paper on [Cut-sweeping](#) added, a light-weight alternative to SAT-sweeping useful for preprocessing AIGs before applying SAT.
- Added slides for invited talk given by Niklas Een at FMCAD ([Linux/Cygwin/Windows](#)).
- Paper on [efficient CNF generation](#) added ([slides](#) also available).
- Bug-fix in MiniSat+ for trivially unsatisfiable instances.
- MiniSat v.2.0 has been released.
- Bug-fix to the proof-logging version of MiniSat (thanks to Georg Weissenbacher for reporting the problem!).
- Bug-fix to the C-version of MiniSat.
- Paper on MiniSat+ added.
- Removed version 1.13 of MiniSat. It was an intermediate version that apparently was buggy.
- Patch for Visual Studio users added.
- Buggy v1.13 Cygwin binary replaced with working v1.14 binary. The bug did not affect the Linux version.
- Fixed the output of MiniSat+ so that it is flushed properly when stdout is redirected to a file.

Building MiniSAT

- Ubuntu (and probably other distros): download minisat-2.2.0.tar.gz, unzip, follow instructions.
- OS X: some portability patches required, unfortunately not very centrally documented. Instead, you can do `brew install minisat`.
- Windows: avoid for the purpose of coursework, as it requires cygwin. There are pre-built binaries but they are old (ver 1.14).

Minisat

- Every line starting with `c` contains comment.
- You should define the problem size with:
 - `p cnf [number of literals] [number of clauses]`
- Subsequent lines contain each individual conjunct; literals are numbers, negations are `-` signs. Each line should end with `0`.

$$(x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_1 \vee x_5 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee x_4)$$

c blah blah...

p cnf 5 3

1 -5 4 0

-1 5 3 4 0

-3 4 0

"test.in"

c blah blah...

p cnf 5 3

1 -5 4 0

-1 5 3 4 0

-3 4 0

\$ minisat test.in

=====[Problem Statistics]=====

Number of variables:	5
Number of clauses:	3
Parse time:	0.00 s
Eliminated clauses:	0.00 Mb
Simplification time:	0.00 s

=====[Search Statistics]=====

Conflicts	ORIGINAL			LEARNT			Progress
	Vars	Clauses	Literals	Limit	Clauses	Lit/Cl	

=====

```
restarts          : 1
conflicts         : 0          (0 /sec)
decisions         : 1          (0.00 % random) (1133 /sec)
propagations      : 0          (0 /sec)
conflict literals : 0          ( nan % deleted)
Memory used       : 0.17 MB
CPU time          : 0.000883 s
```

SATISFIABLE

\$ _

$$(x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_1 \vee x_5 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee x_4)$$

```
"test.in"
c blah blah...
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 4 0
```

```
$ minisat test.in test.out
...
$ cat test.out
SAT
-1 -2 -3 4 -5 0
$ _
```

```
"test.in"
c blah blah...
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 4 0
1 2 3 -4 5 0
```

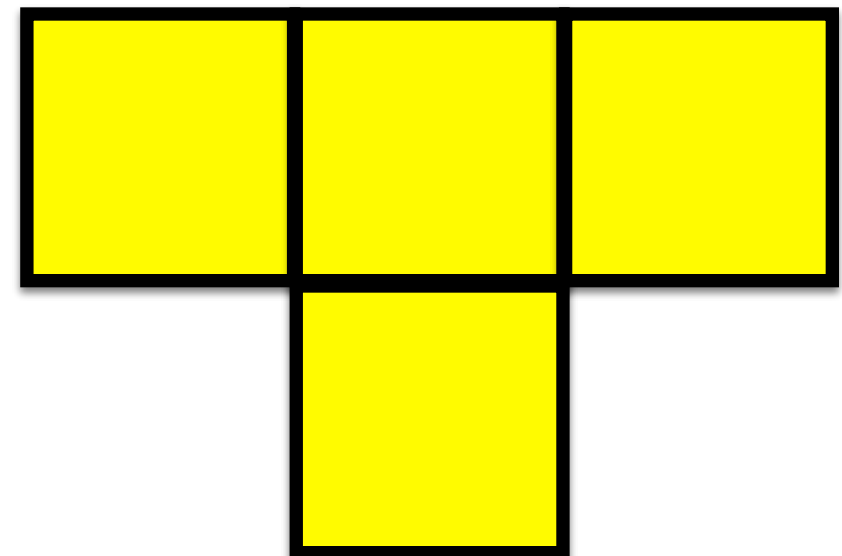
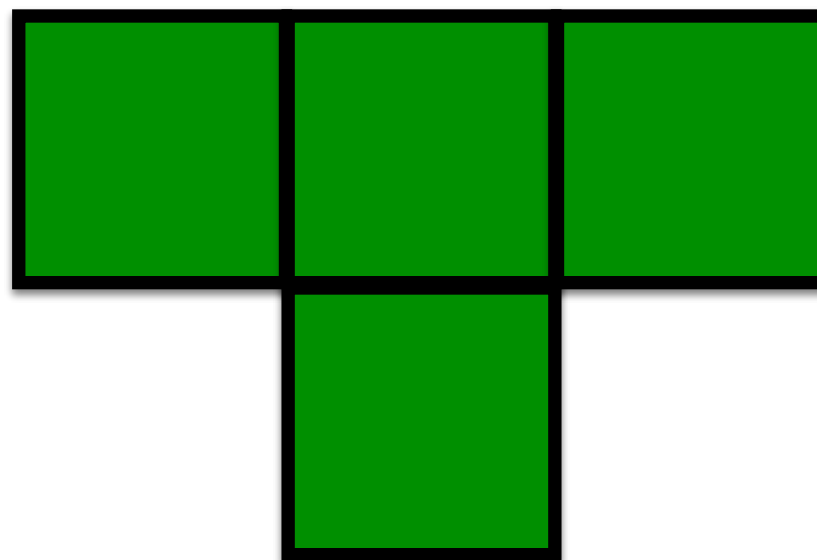
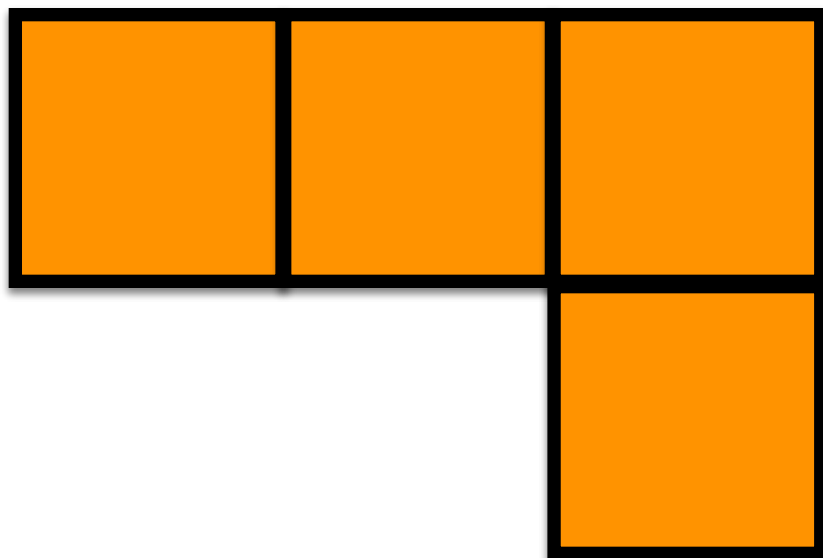
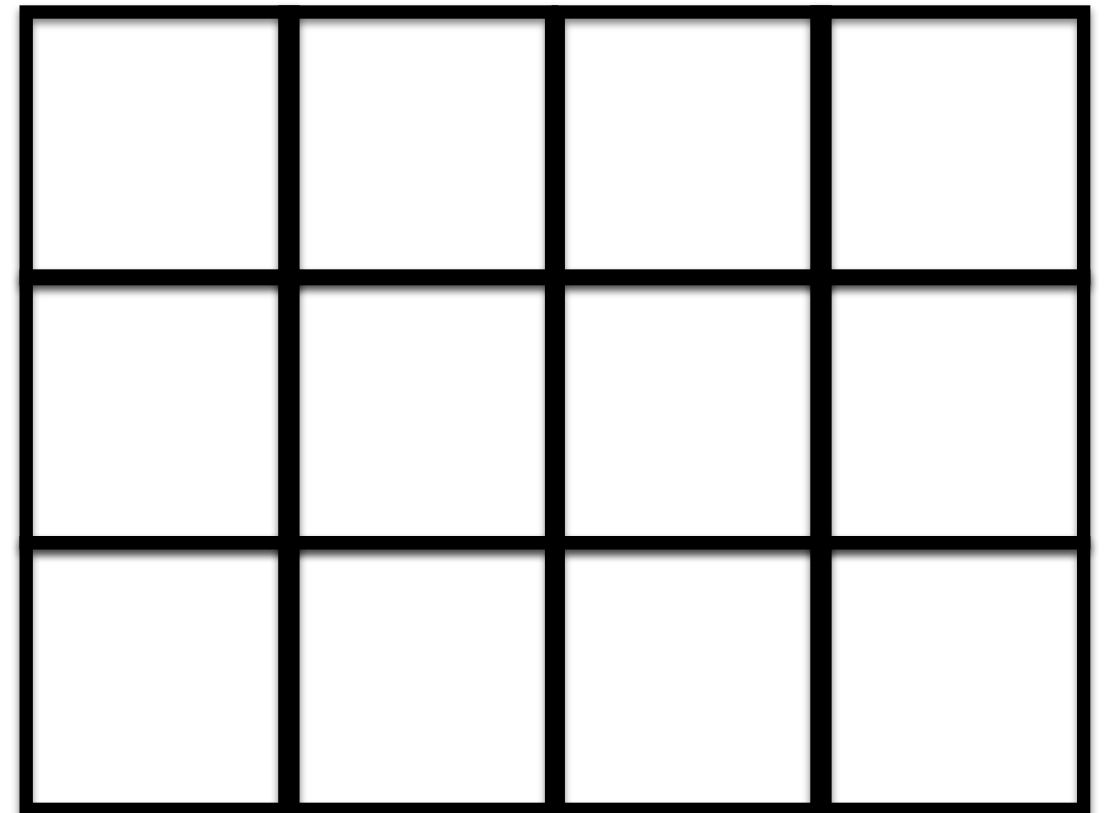
```
$ minisat test.in test.out
...
$ cat test.out
SAT
-1 2 -3 4 -5 0
$ _
```

SAT Encoding

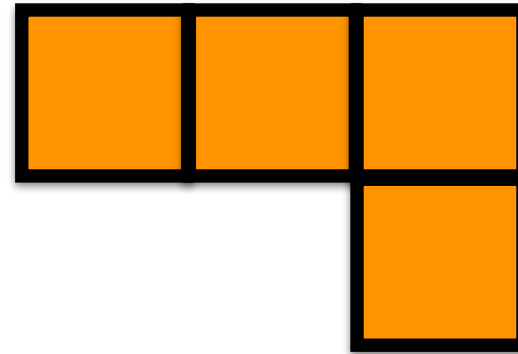
- Formulate a problem into SAT
- Solve the SAT
- Interpret the result back to the original domain

A Tetris-like Puzzle

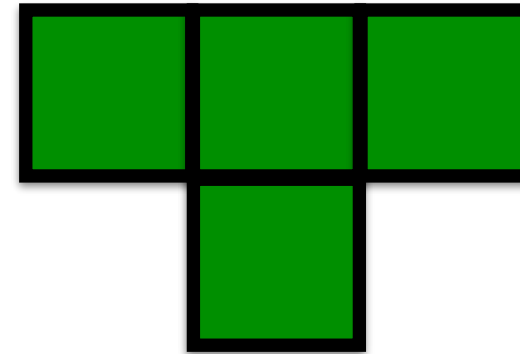
- Fill in the 4 by 3 grid using the given pieces.
- SAT...?



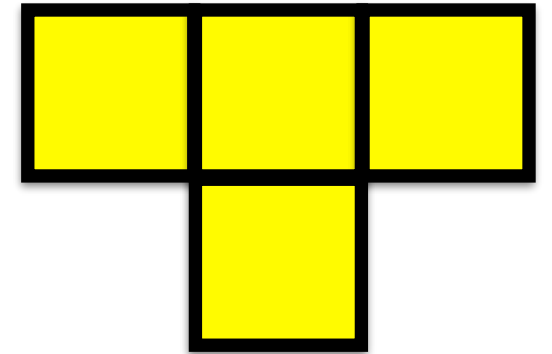
(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)



1



2

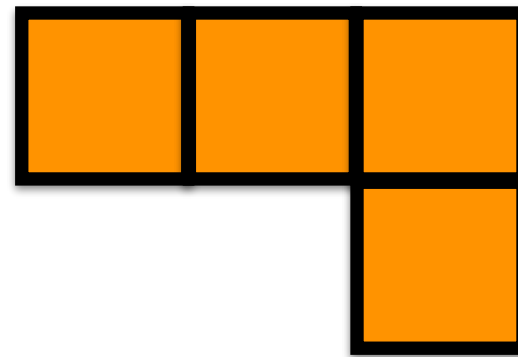


3

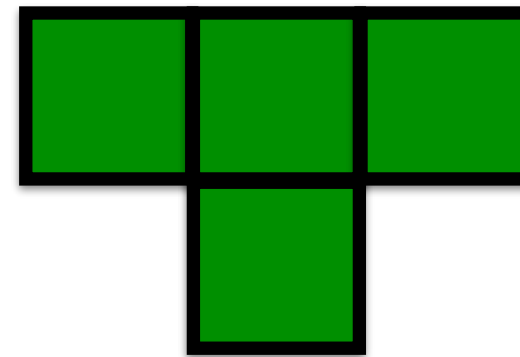
A correct solution should satisfy the following:

- Every grid has at least one piece on it.
- Every grid has at most one piece on it.
- Every piece is used at least once.
- Pieces cannot be broken.

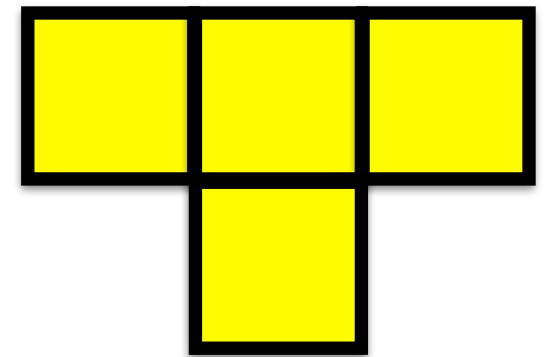
(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)



1



2



3

Let x_{ijk} be the proposition that grid (i, j) .

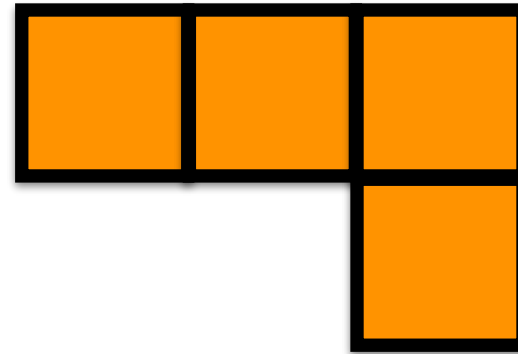
First condition: “every grid has at least one piece on it”.

That is:

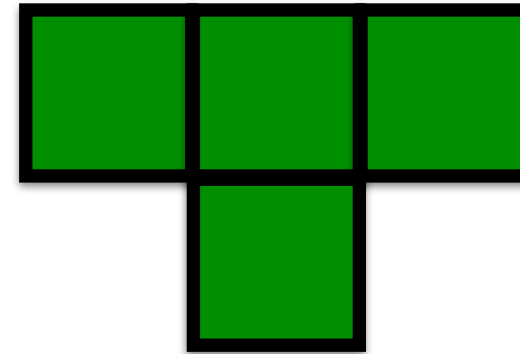
1. for grid (1, 1): $x_{111} \vee x_{112} \vee x_{113}$
2. for grid (1, 2): $x_{121} \vee x_{122} \vee x_{123}$
3. ...

$$\therefore (x_{111} \vee x_{112} \vee x_{113}) \wedge (x_{121} \vee x_{122} \vee x_{123}) \wedge \dots$$

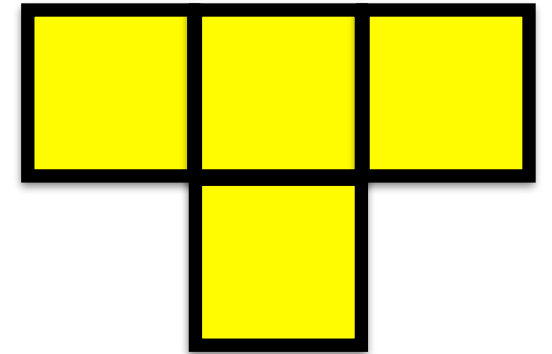
(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)



1



2



3

Second condition: “every grid has at most one piece on it.”

That is, a single grid cannot be shared by two pieces. For example, grid (1, 1) cannot be shared by any pair of pieces.

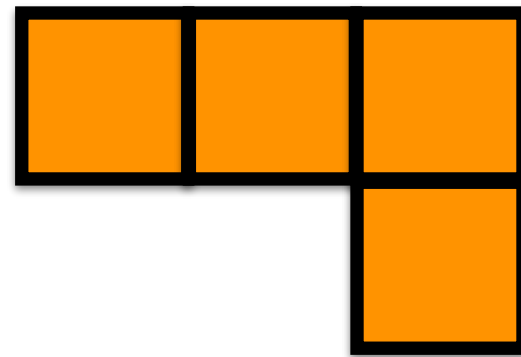
$$1. \neg(x_{111} \wedge x_{112}) = \neg x_{111} \vee \neg x_{112}$$

$$2. \neg(x_{111} \wedge x_{113}) = \neg x_{111} \vee \neg x_{113}$$

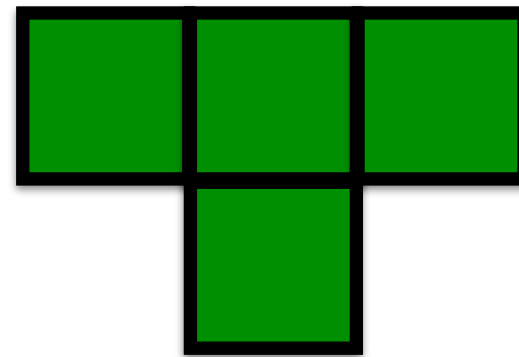
$$3. \neg(x_{112} \wedge x_{113}) = \neg x_{112} \vee \neg x_{113}$$

$$\bigwedge_{(i,j)} \bigwedge_{n \neq m} \neg(x_{ijn} \wedge x_{ijm}) = \bigwedge_{(i,j)} \bigwedge_{n \neq m} (\neg x_{ijn} \vee \neg x_{ijm})$$

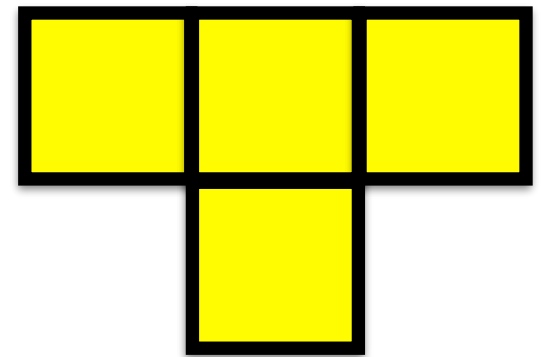
(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)



1



2



3

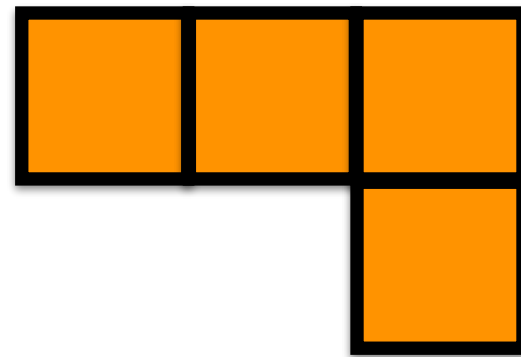
Third condition: “every piece is used at least once.”

One possible encoding: for a piece with n squares, specify that every combination of $n + 1$ grids being assigned with that symbol evaluates to *false*. But this explodes in size.

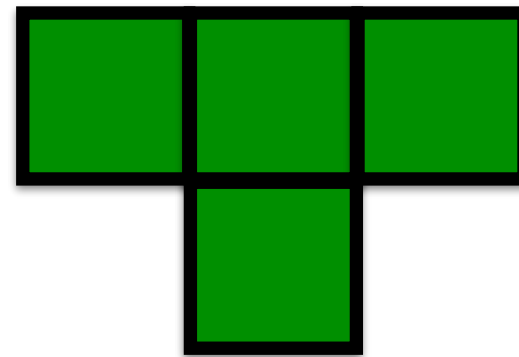
Since the whole grids will be filled when all pieces are used just once, we can restate the condition as the following: for each piece, at least one of the grids is assigned to it. For example, $x_{111} \vee x_{121} \vee x_{131} \vee \dots \vee x_{341}$. To combine for n pieces:

$$\bigwedge_n \bigvee_{(i,j)} x_{ijn}$$

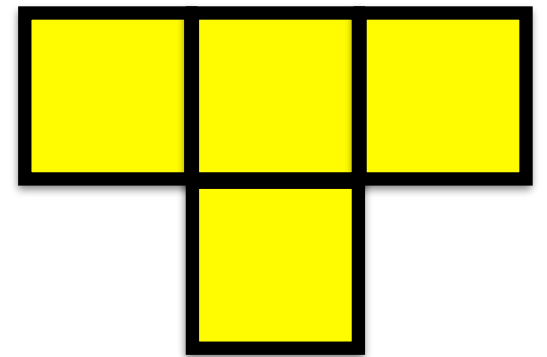
(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)



1



2



3

Fourth condition: “pieces cannot be broken”.

That is, for piece 1, either $x_{111} \wedge x_{121} \wedge x_{131} \wedge x_{231}$ or
 $x_{121} \wedge x_{131} \wedge x_{141} \wedge x_{241}$ or
 $x_{211} \wedge x_{221} \wedge x_{231} \wedge x_{331}$ or ...

Unfortunately, this part is NOT in CNF.

So I cheated a bit; instead of `minisat`, I turned to `z3`.

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

3	2	2	2
3	3	2	1
3	1	1	1

```
(define-fun x20 () Bool true) 2,3,2
(define-fun x5  () Bool true) 1,2,2
(define-fun x8  () Bool true) 1,3,2
(define-fun x11 () Bool true) 1,4,2
```

```
(define-fun x3  () Bool true) 1,1,3
(define-fun x15 () Bool true) 2,1,3
(define-fun x27 () Bool true) 3,1,3
(define-fun x18 () Bool true) 2,2,3
```

```
(define-fun x22 () Bool true) 2,4,1
(define-fun x28 () Bool true) 3,2,1
(define-fun x31 () Bool true) 3,3,1
(define-fun x34 () Bool true) 3,4,1
```

<https://github.com/Z3Prover/z3>

<http://rise4fun.com/z3/tutorial>