MiniSAT and SAT Encoding

CS402, Spring 2017

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.

$$l_1 \lor \ldots \lor l_n \text{ can be rewritten into 3-SAT as}$$
$$(l_1 \lor l_2 \lor x_2) \land (\neg x_2 \lor l_3 \lor x_3) \land (\neg x_3 \lor l_4 \lor x_4) \land \ldots$$
$$\land (\neg x_{n-3} \lor l_{n-2} \lor x_{n-2}) \land (\neg x_{n-2} \lor l_{n-1} \lor l_n)$$

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

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
```

```
“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
$ _
```

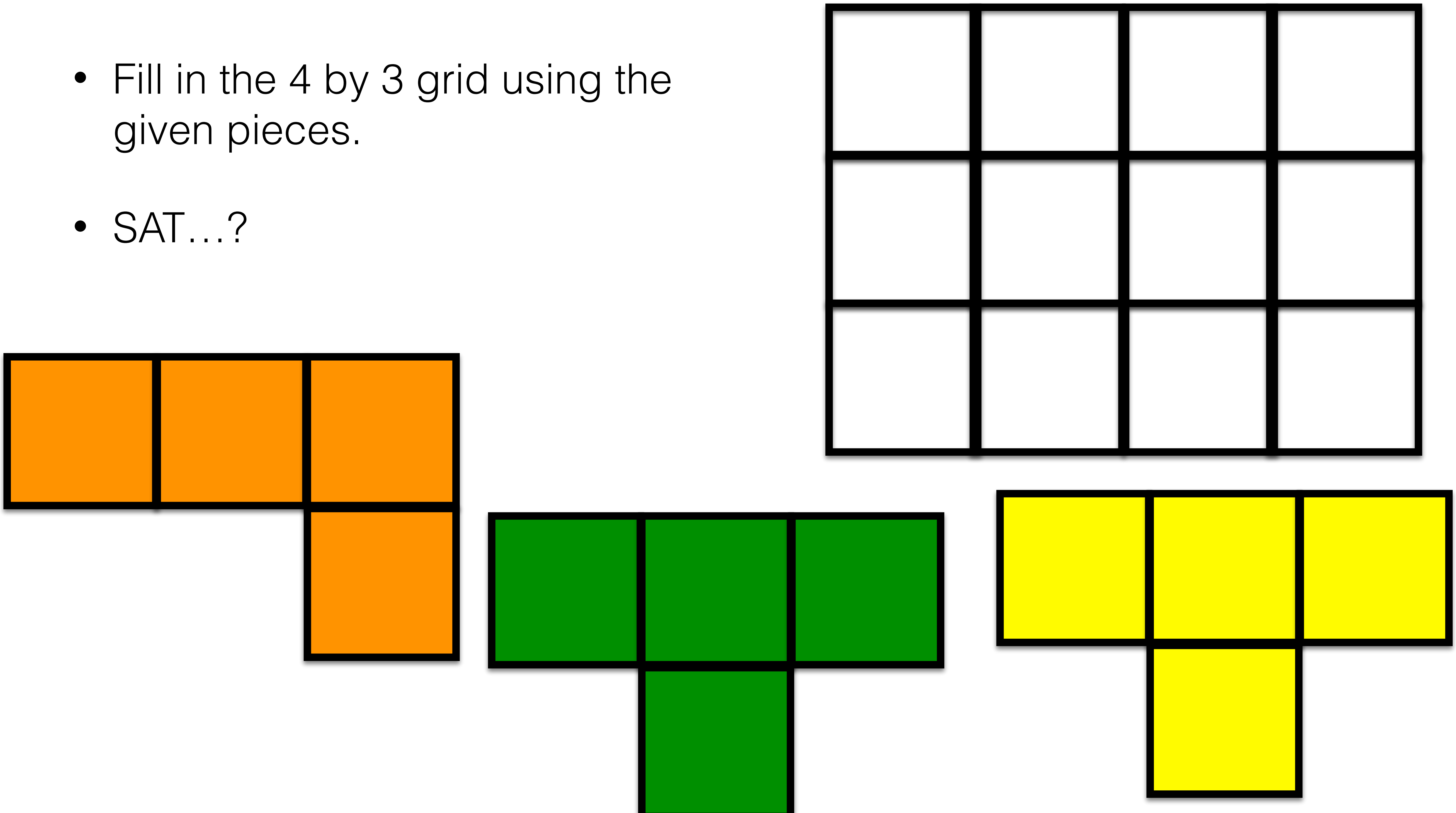
```
$ minisat test.in test.out
…
$ cat test.out
SAT
-1 2 -3 4 -5 0
$ _
```

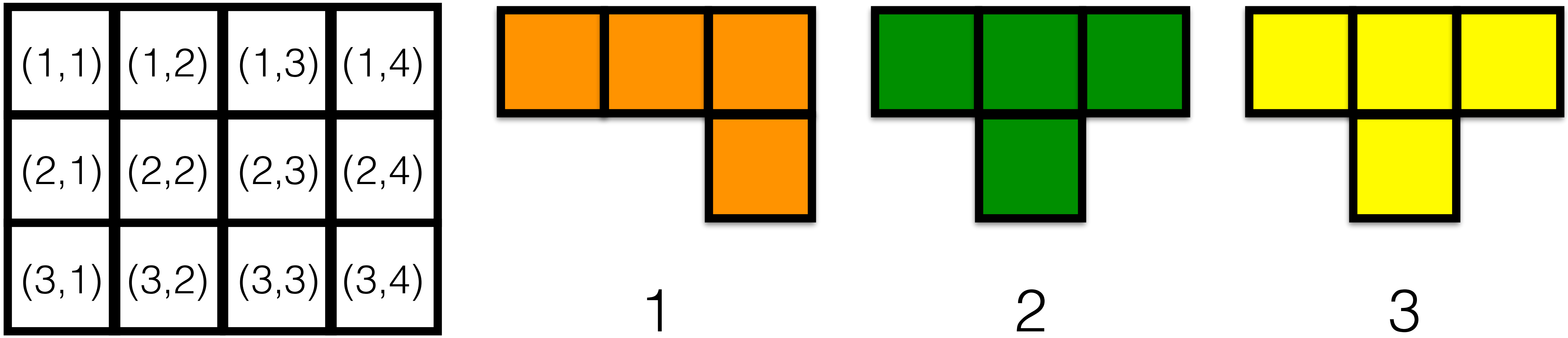
SAT Encoding

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

A Tetris-like Puzzle

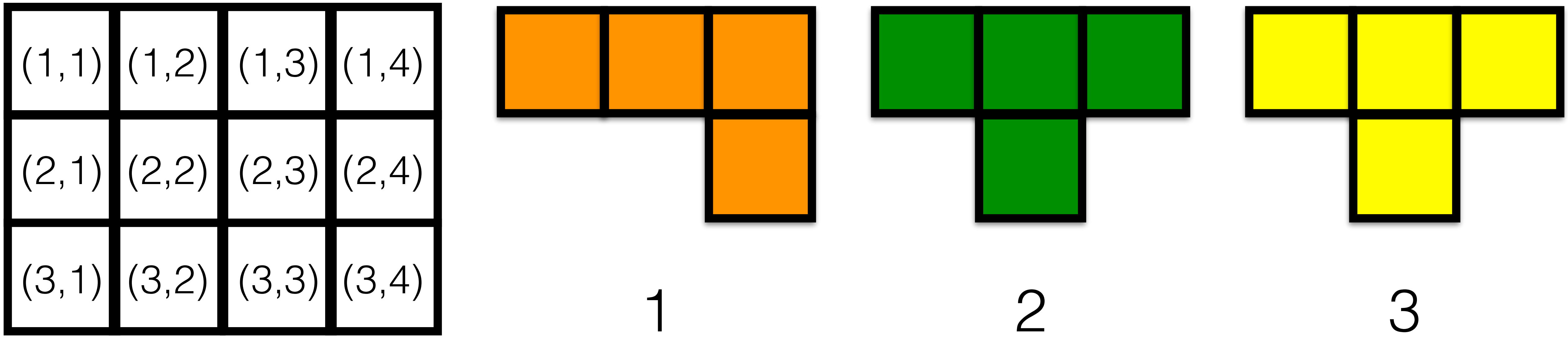
- Fill in the 4 by 3 grid using the given pieces.
- SAT…?





A correct solution should satisfy the following:

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

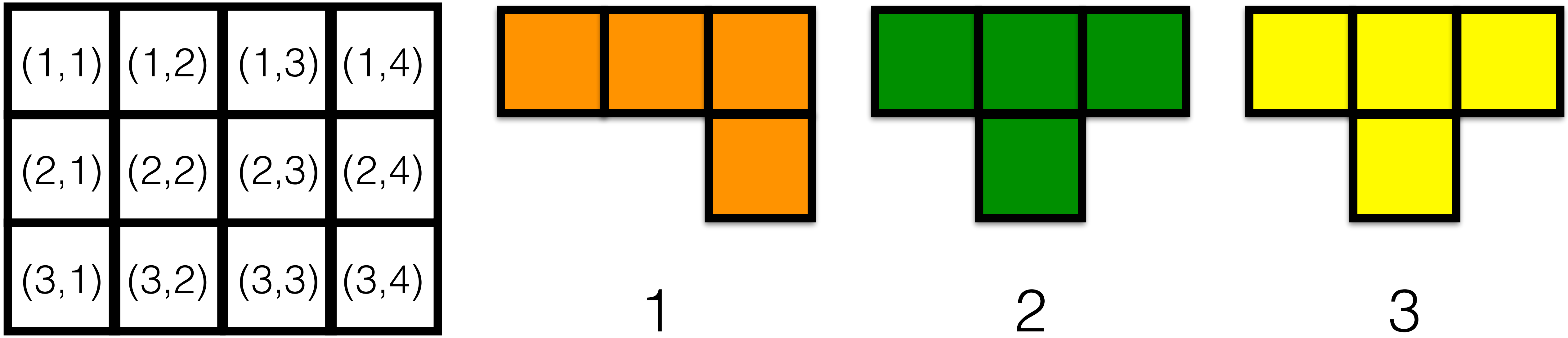


Let x_{ijk} be the proposition that grid (i, j).
First condition: “every grid has at least one piece on it”.

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. $\ldots$

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

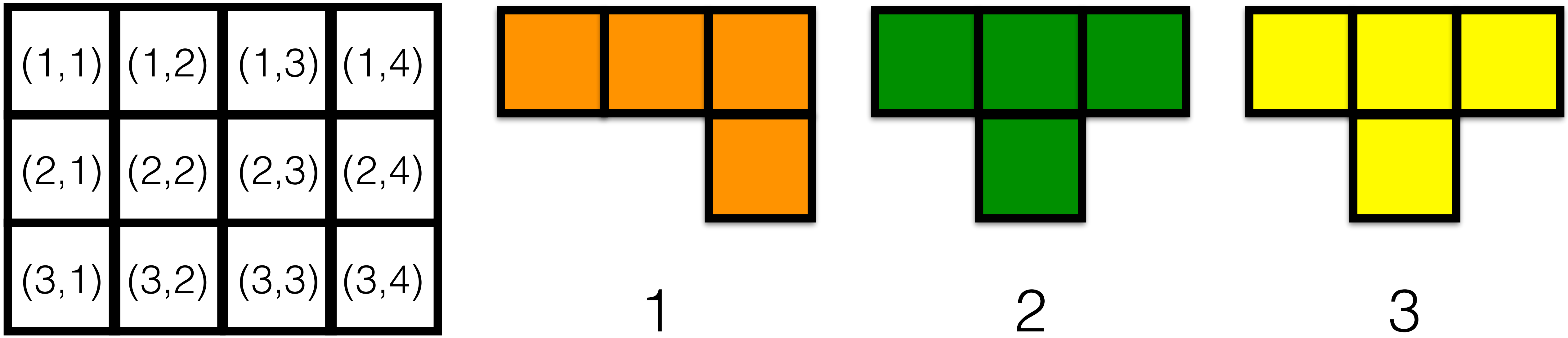


Second condition: “every grid has at most one piece on it.”

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

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

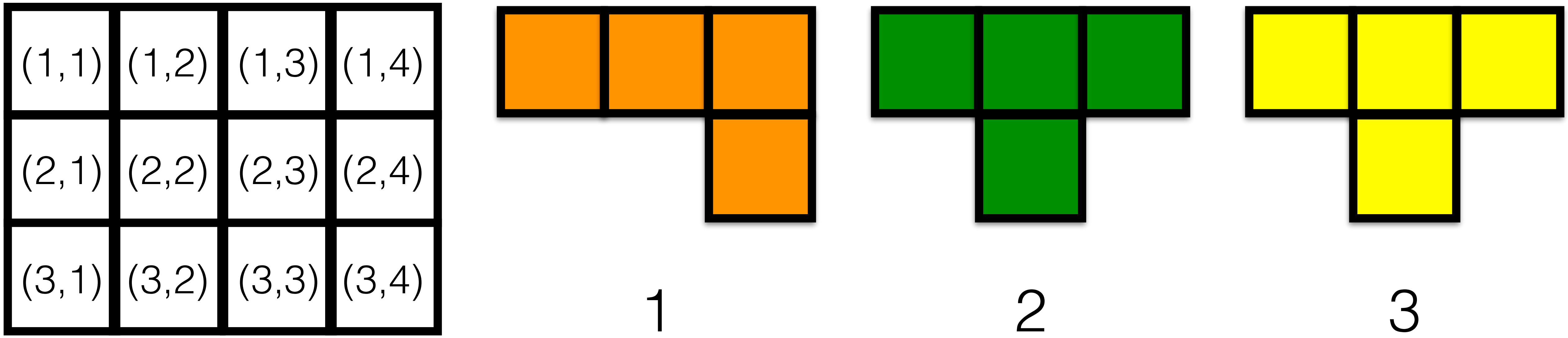


Third condition: "every piece is used at least once."

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, $x_{111} \vee x_{121} \vee x_{131} \vee \ldots \vee x_{341}$. To combine for n pieces:

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



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)

```
(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
```

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

https://github.com/Z3Prover/z3

http://rise4fun.com/z3/tutorial