MiniCP

Constraint Programming

Table Constraint

Hélène Verhaeghe & Pierre Schaus

Table Constraint

x	y	z
1	2	3
1	3	3
2	1	3
2	1	1
3	3	3
4	1	2
4	4	4

A table constraint has an enumeration of the possible assignments for its variables (here x, y, and z).

Semantics

$(x=1 \wedge y=2 \wedge z=3) \vee (x=1 \wedge y=3 \wedge z=3) \vee (x=2 \wedge y=1 \wedge z=3) \vee \ldots$

Signature

```
/**
 * Fixing x_0 = v_0, x_1 = v_1, … is only
 * valid if there exists a row (v_0, v_1, ...) in table.
 */
public Table(IntVar[] x, int[][] table)
```

Intensional vs Extensional Formulation

- A constraint like AllDifferent([x,y,z]) is said to be intensional. The solution set to the constraint is *implicit* with the semantics of the constraint.

- To make it *explicit*, via an extensional formulation, aka a Table constraint, we list *all* the solutions. For D(x) = D(y) = D(z) = {0..2} we have

x	y	z
0	1	2
0	2	1
1	0	2
1	2	0
2	0	1
2	1	0

- For n or n–1 variables over a domain of size n, the extensional formulation of AllDifferent requires n! tuples.
  That is why intensional formulations are interesting.

- An extensional formulation can impose *any* relation on its variables.
  That is why Table constraints are interesting:
  this is the most flexible form of constraints.

If an efficient intensional constraint with a domain-consistent filtering exists in your CP solver, then you should probably prefer to use it rather than an extensional formulation of the problem constraint.

Most flexible constraint of the universe

- Any predicate on k variables can be turned into a table constraint
- Just enumerate the solutions to the constraint into a table

X+Y=Z

X	Y	Z
0	0	0
0	1	1
0	2	2
1	0	1
1	1	2
1	2	3
2	0	2
2	1	3
2	2	4

**if X is even, then Y=2,
else Z>0**

X	Y	Z
0	2	0
0	2	1
0	2	2
1	0	1
1	0	2
1	1	1
1	1	2
1	2	1
1	2	2
2	2	0
2	2	1
2	2	2

AllDifferent(X,Y,Z)

X	Y	Z
0	1	2
0	2	1
1	0	2
1	2	0
2	0	1
2	1	0

Most flexible constraint of the universe

- A practical example is the solving of the Enigma machine.
- Given an output and tiny clues about the input, can we find the full input and the settings of the Enigma machine?
- Yes! Using the Modulo constraint X mod Y = Z.

Antuori V., Portoleau T., Rivière L. and Hébrard E.
On How Turing and Singleton Arc Consistency Broke the Enigma Code.
CP 2021.

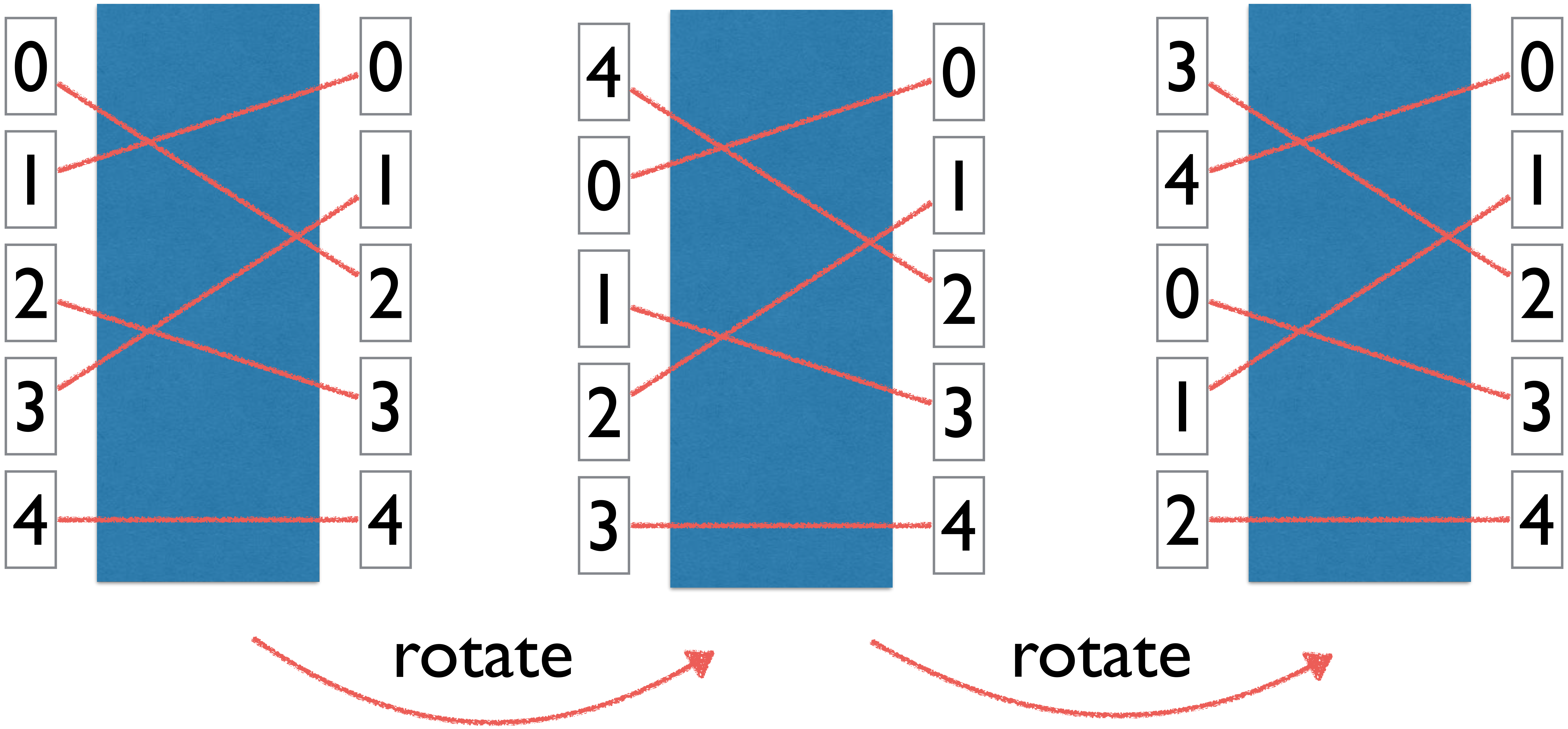
Application: Enigma machine

- Composed of rotors, input = [0,3,4], encoded as [2,4,0]
- After each input, the rotor rotates by one position

- Step 0
- Input 0
- Output 2

- Step 1
- Input 3
- Output 4

- Step 2
- Input 4
- Output 0



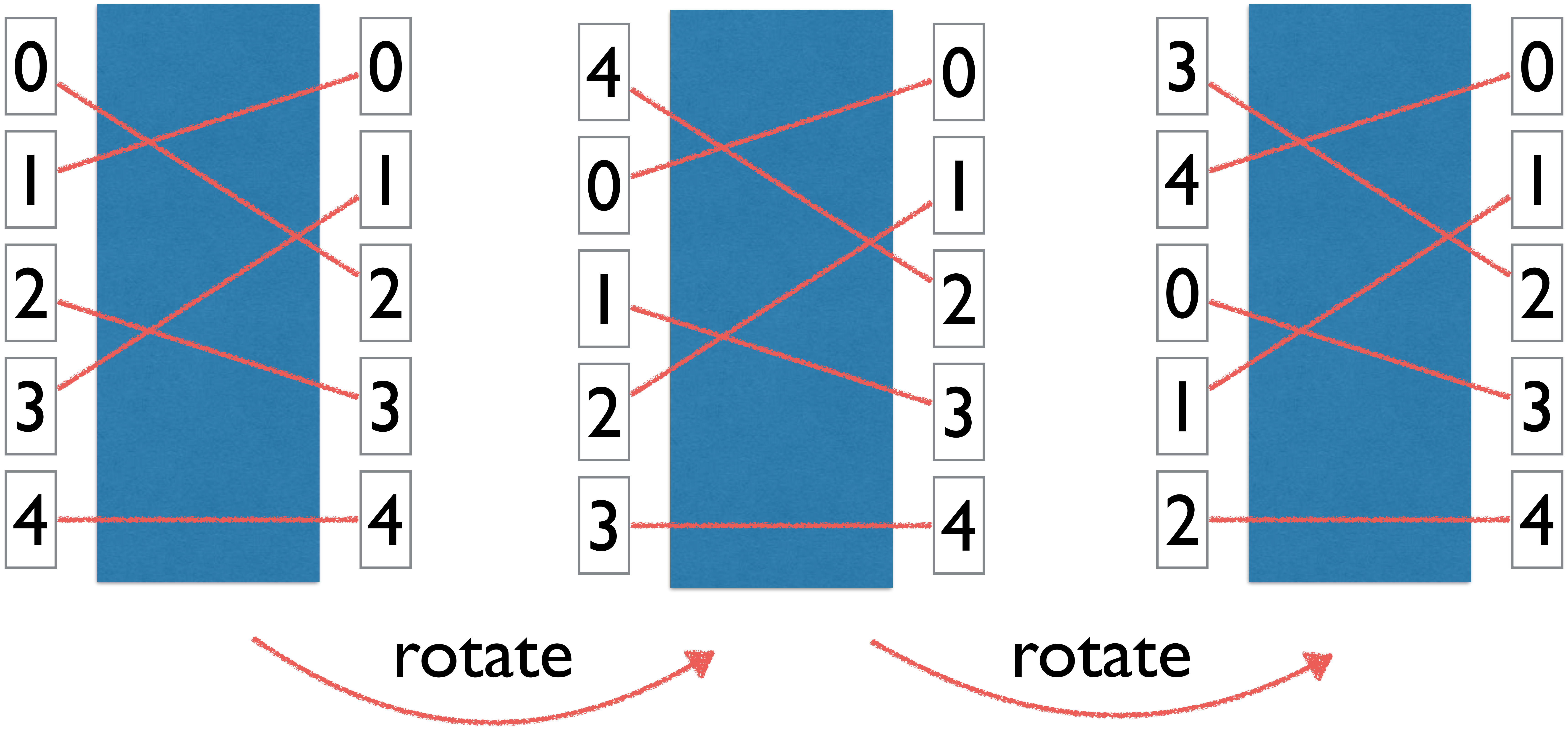
Application: Enigma machine

- T = [2,0,3,1,4] (mapping at initial position)
- What is the output at step i for input I? Answer: output = T[(I+i)%5]

- Step 0
- Input 0
- Output 2

- Step 1
- Input 3
- Output 4

- Step 2
- Input 4
- Output 0



Application: Enigma machine

- T = [2,0,3,1,4] (mapping at initial position)
- What is the output at step i for input I? Answer: output = T[(I+i)%5] = A

I	i	A
0	0	2
0	1	0
0	2	3
0	3	1
0	4	4
0	5	2
0	6	0
0	7	3
0	8	1
0	9	4
0	10	2
0	11	0
0	12	3
0	13	1
0	14	4
...	...	...

I	i	A
1	0	0
1	1	3
1	2	1
1	3	4
1	4	2
1	5	0
1	6	3
1	7	1
1	8	4
1	9	2
1	10	0
1	11	3
1	12	1
1	13	4
1	14	2
...	...	...

I	i	A
2	0	3
2	1	1
2	2	4
2	3	2
2	4	0
2	5	3
2	6	1
2	7	4
2	8	2
2	9	0
2	10	3
2	11	1
2	12	4
2	13	2
2	14	0
...	...	...

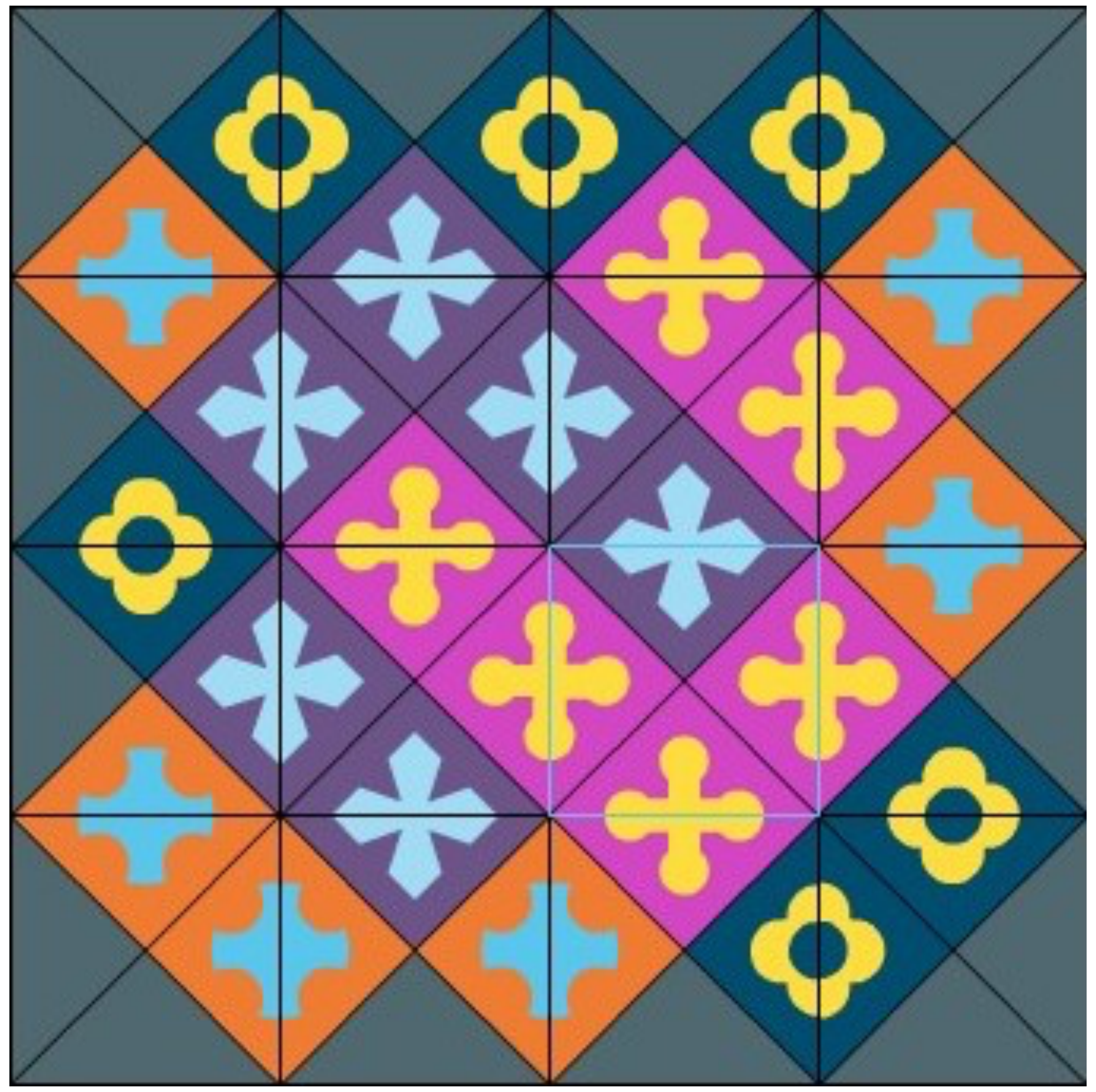
I	i	A
3	0	1
3	1	4
3	2	2
3	3	0
3	4	3
3	5	1
3	6	4
3	7	2
3	8	0
3	9	3
3	10	1
3	11	4
3	12	2
3	13	0
3	14	3
...	...	...

I	i	A
4	0	4
4	1	2
4	2	0
4	3	3
4	4	1
4	5	4
4	6	2
4	7	0
4	8	3
4	9	1
4	10	4
4	11	2
4	12	0
4	13	3
4	14	1
...	...	...

Application of Table constraints: Eternity Puzzle

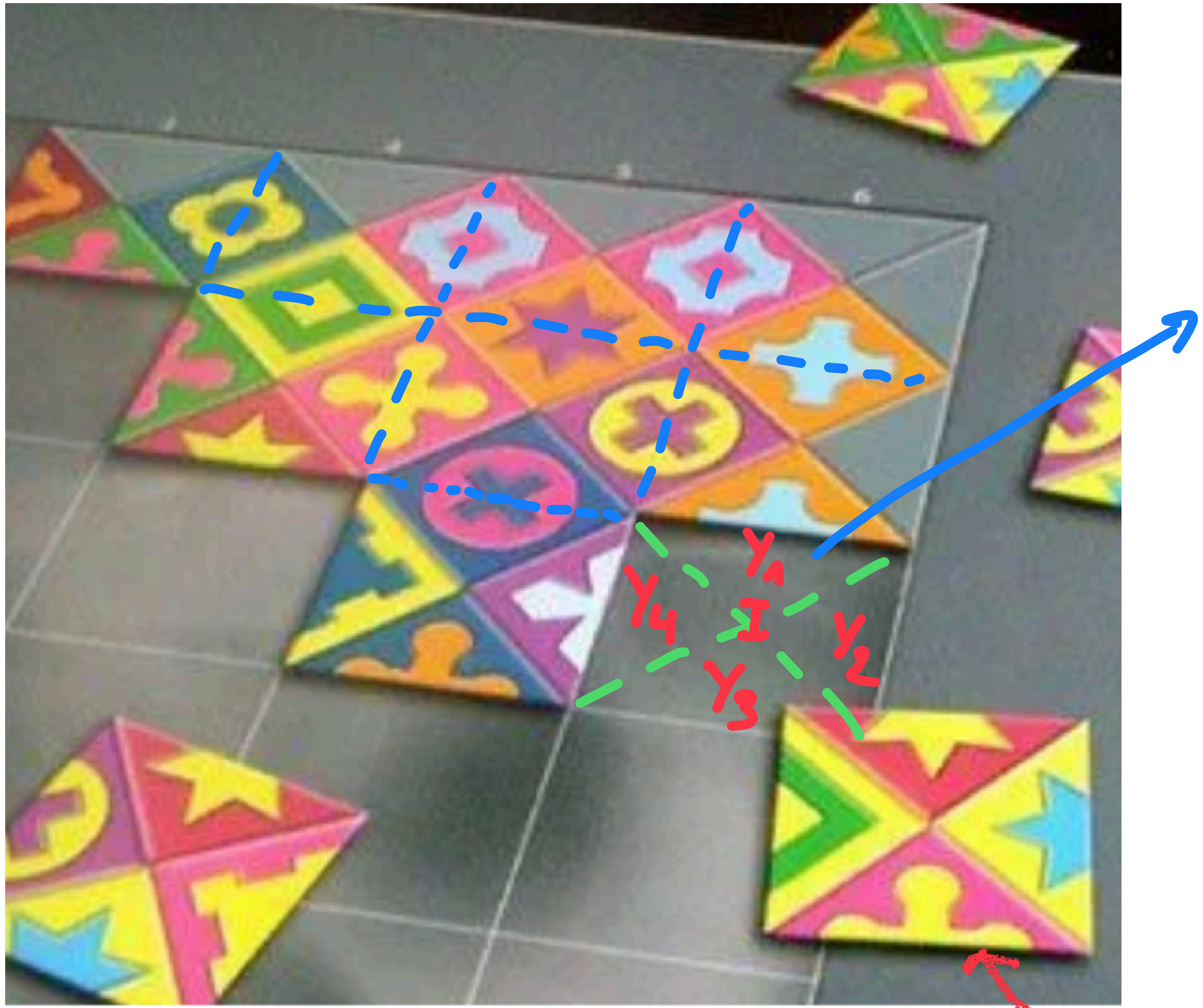
Eternity II Puzzle

- Edge matching puzzle: place 256 square pieces into a 16x16 grid, constrained by the requirement to match adjacent edges.



- How to model this puzzle?
- https://en.wikipedia.org/wiki/Eternity_II_puzzle

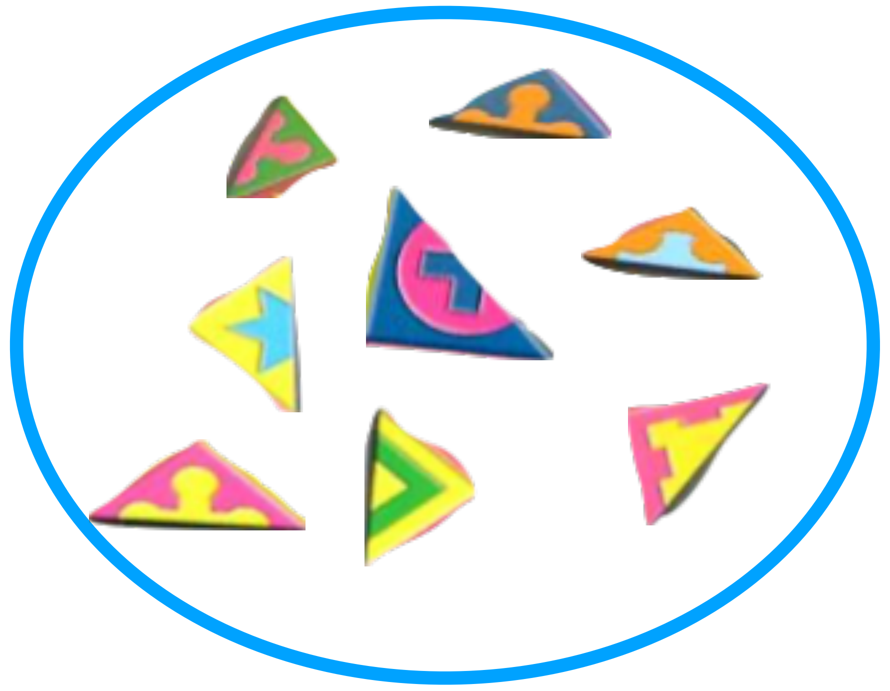
Decision variables for Eternity II



For each position, we need 5 variables:

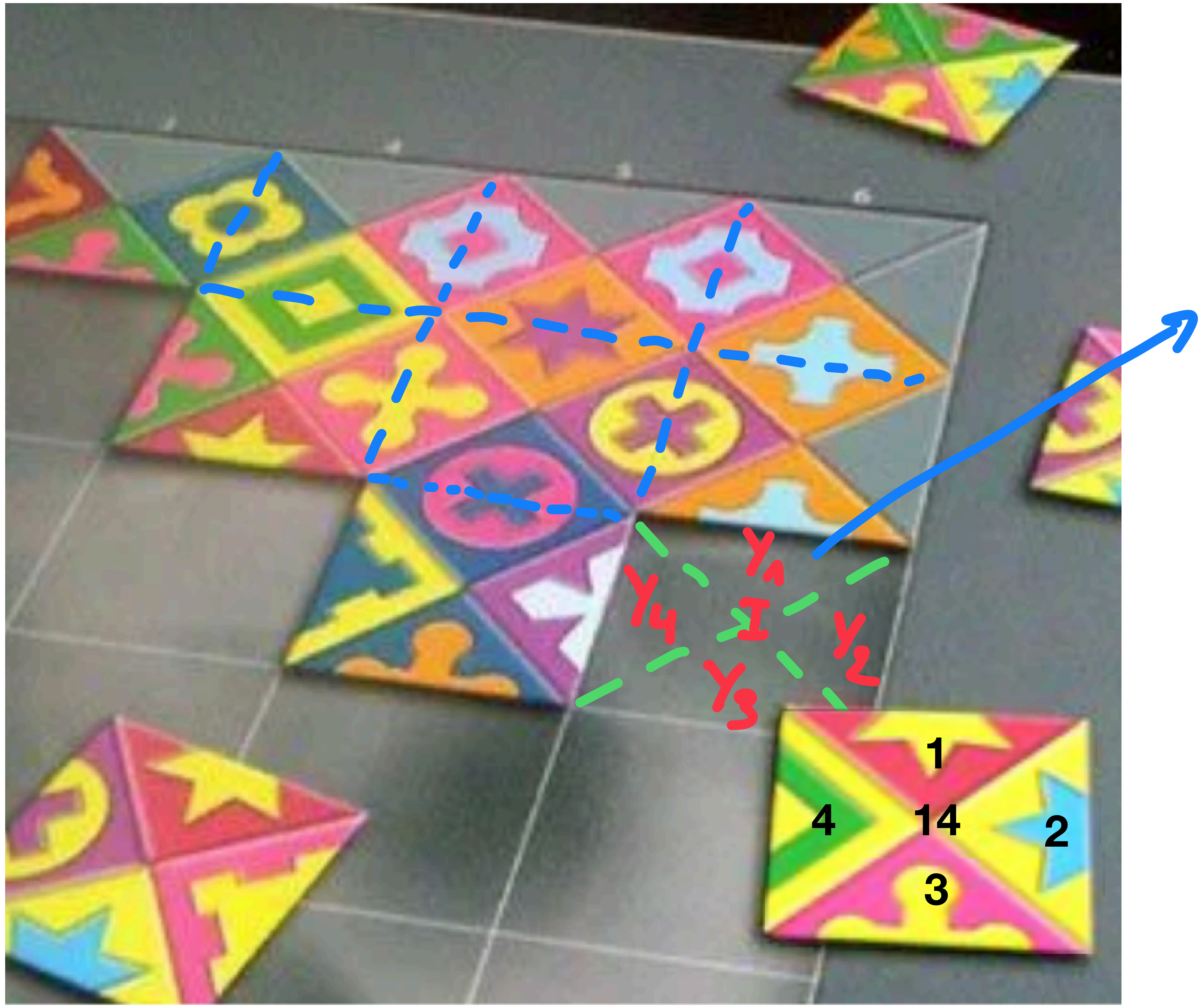
- 1 variable S_i (but Y_i on the picture) for each of the 4 sides
- 1 variable I for the identifier of the placed piece

$D(S_i)$ =



How do we model that (I,S1,S2,S3,S4) corresponds to a valid piece of the game, such as this one ?

Decision variables for Eternity II



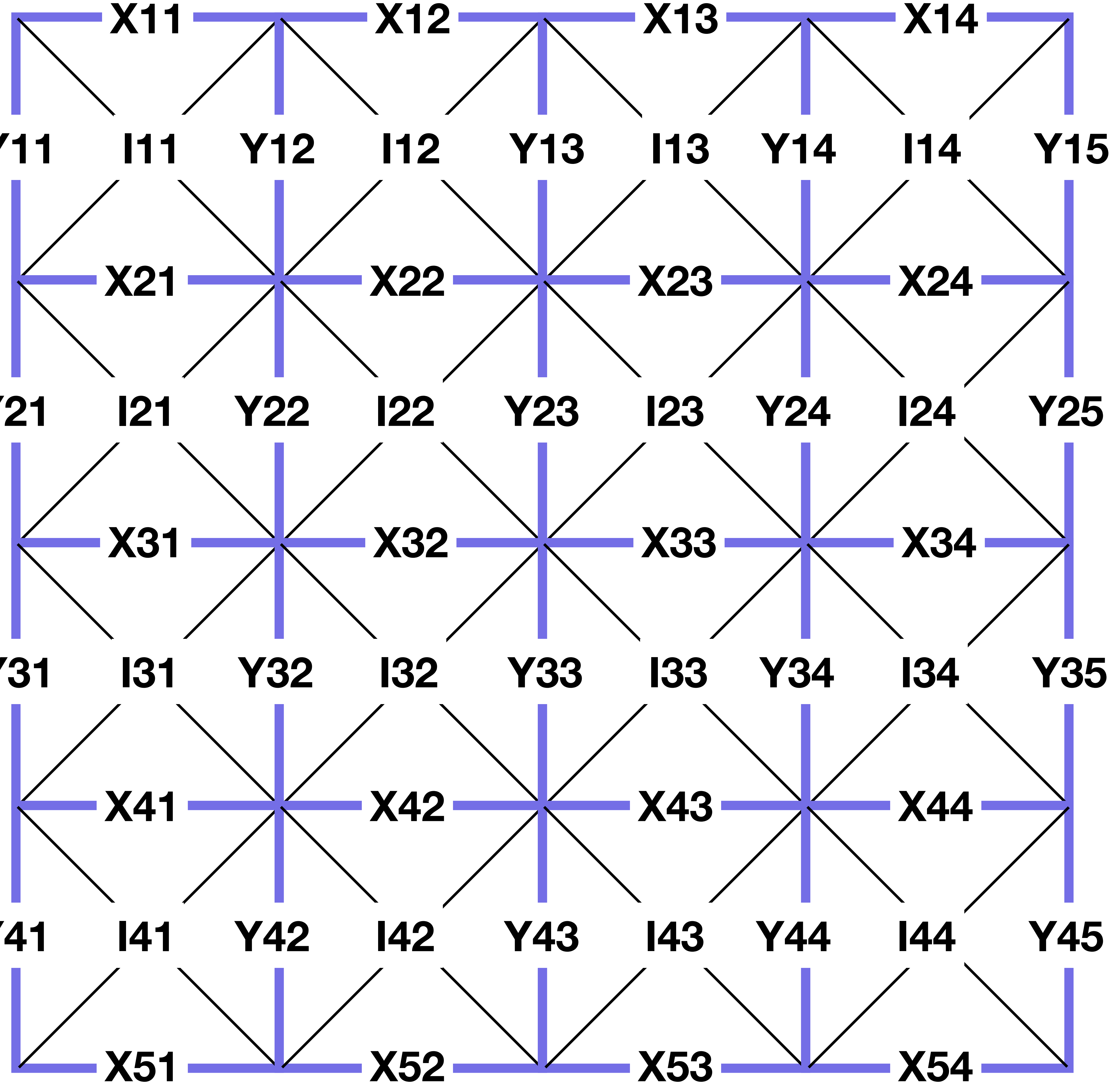
Answer: With a Table constraint!
Let us build it together

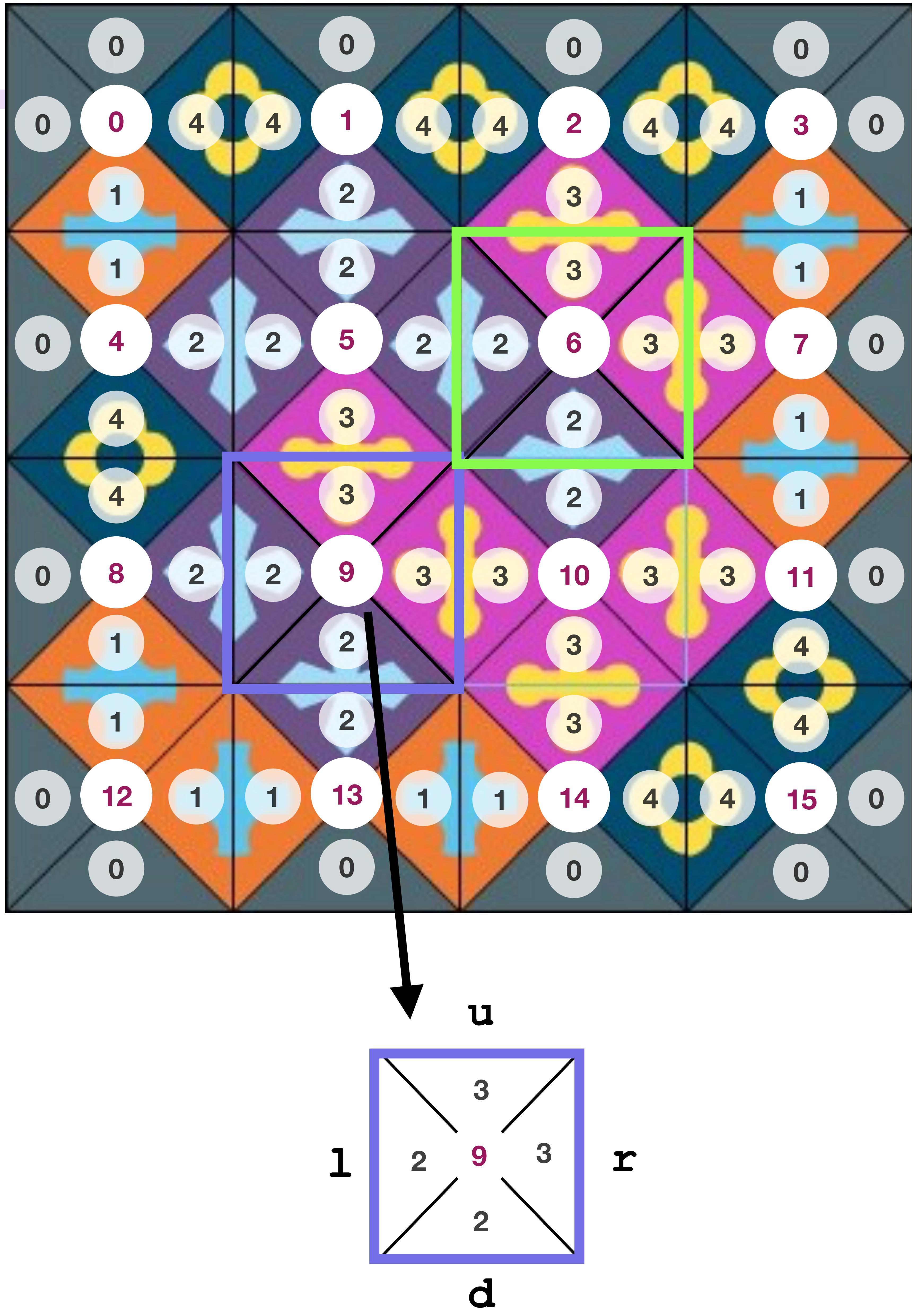
I	S1	S2	S3	S4
14	1	2	3	4
14	2	3	4	1
14	3	4	1	2
14	4	1	2	3

(I,S1,S2,S3,S4) ∈ table ensures that it corresponds to one of the four rotations for this piece

Model for 4x4 Eternity II

D(X21)={0..4} D(I11)={0..15}


X11 X12 X13 X14
Y11 I11 Y12 I12 Y13 I13 Y14 I14 Y15
X21 X22 X23 X24
Y21 I21 Y22 I22 Y23 I23 Y24 I24 Y25
X31 X32 X33 X34
Y31 I31 Y32 I32 Y33 I33 Y34 I34 Y35
X41 X42 X43 X44
Y41 I41 Y42 I42 Y43 I43 Y44 I44 Y45
X51 X52 X53 X54



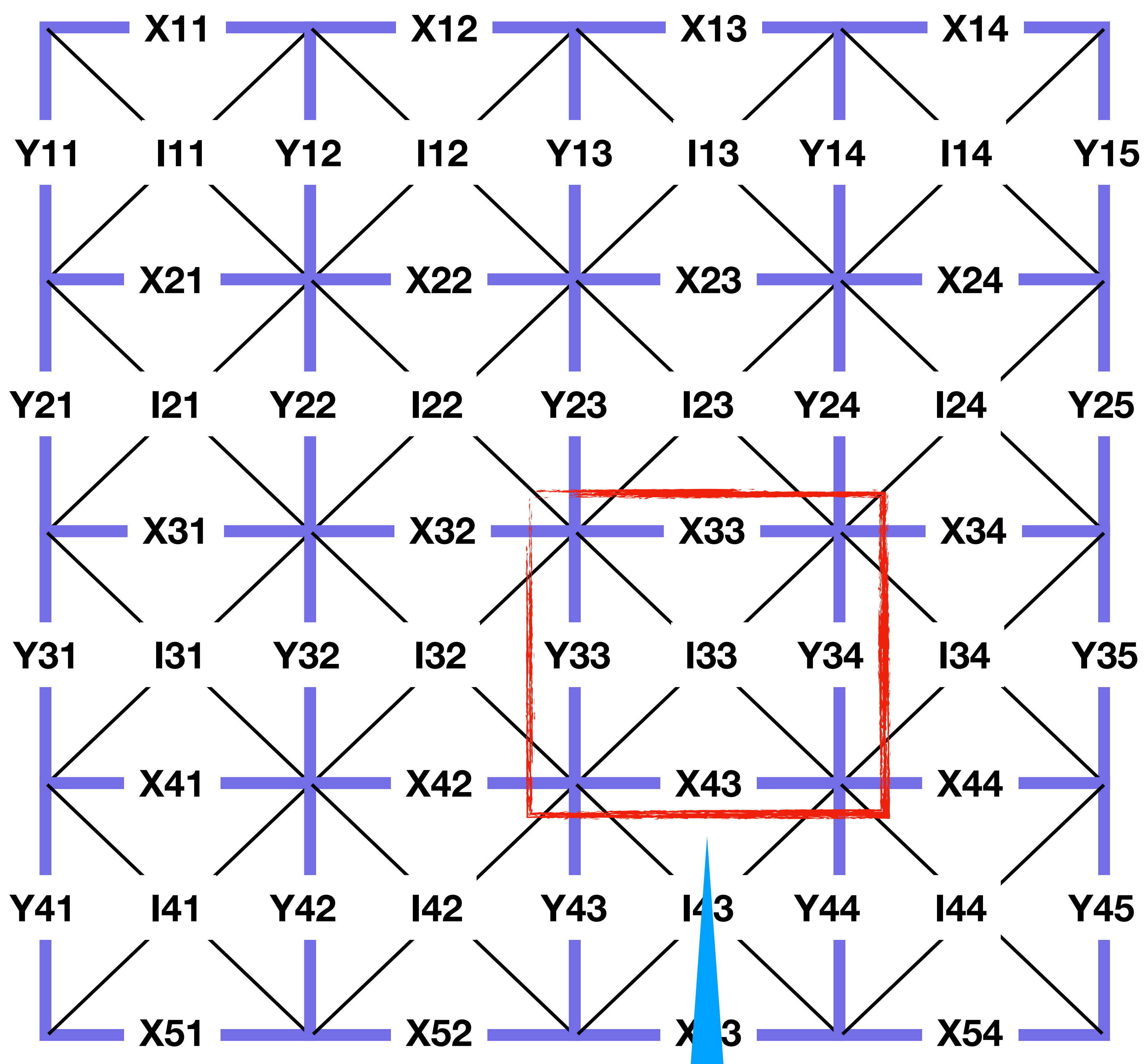
tableOfPieces

i	u	r	d	l
9	3	3	2	2
9	3	2	2	3
9	2	2	3	3
9	2	3	3	2
6	3	3	2	2
6	3	2	2	3
6	2	2	3	3
6	2	3	3	2
...				

Every piece has 4 possible rotations,
hence 4 entries per piece are created

Model for 4x4 Eternity

D(X21)={0..4}   D(I11)={0..15}


X11 X12 X13 X14
Y11 I11 Y12 I12 Y13 I13 Y14 I14 Y15
X21 X22 X23 X24
Y21 I21 Y22 I22 Y23 I23 Y24 I24 Y25
X31 X32 X33 X34
Y31 I31 Y32 I32 Y33 I33 Y34 I34 Y35
X41 X42 X43 X44
Y41 I41 Y42 I42 Y43 I43 Y44 I44 Y45
X51 X52 X53 X54

All the pieces are placed and each can be placed only once:
AllDifferent(I11,I12,…,I44)

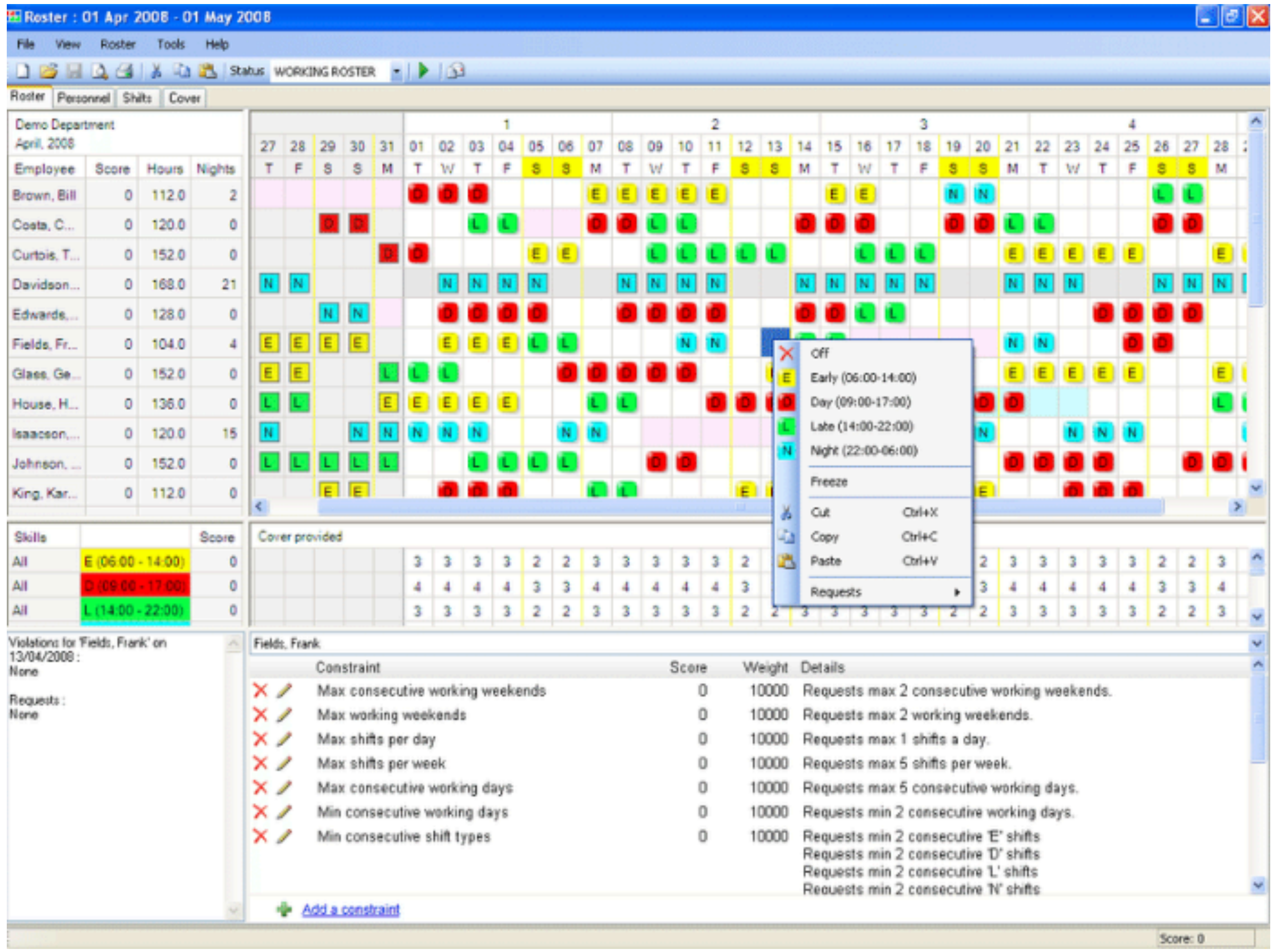
All the positions are occupied with valid pieces
$(I_{ij},X_{i,j},Y_{i,j+1},X_{i+1,j},Y_{ij}) \in$ tableOfPieces $\forall$ i,j $\in$ [1..4]x[1..4]

Each square contains a valid piece:
[I33,X33,Y34,X43,Y33] ∈ tableOfPieces

tableOfPieces

i	u	r	d	l
9	3	3	2	2
9	3	2	2	3
9	2	2	3	3
9	2	3	3	2
6	3	3	2	2
6	3	2	2	3
6	2	2	3	3
6	2	3	3	2
…				

Application of Table constraints: Regular (Automaton) constraint for rostering problems



Demo Department
April, 2008

Employee	Score	Hours	Nights
Brown, Bill	0	112.0	2
Costa, C...	0	120.0	0
Curtois, T...	0	152.0	0
Davidson...	0	168.0	21
Edwards,...	0	128.0	0
Fields, Fr...	0	104.0	4
Glass, Ge...	0	152.0	0
House, H...	0	136.0	0
Isaacson,...	0	120.0	15
Johnson, ...	0	152.0	0
King, Kar...	0	112.0	0

Skills		Score
All	E (06:00 - 14:00)	0
All	D (09:00 - 17:00)	0
All	L (14:00 - 22:00)	0

Cover provided

Violations for 'Fields, Frank' on 13/04/2008 :
None

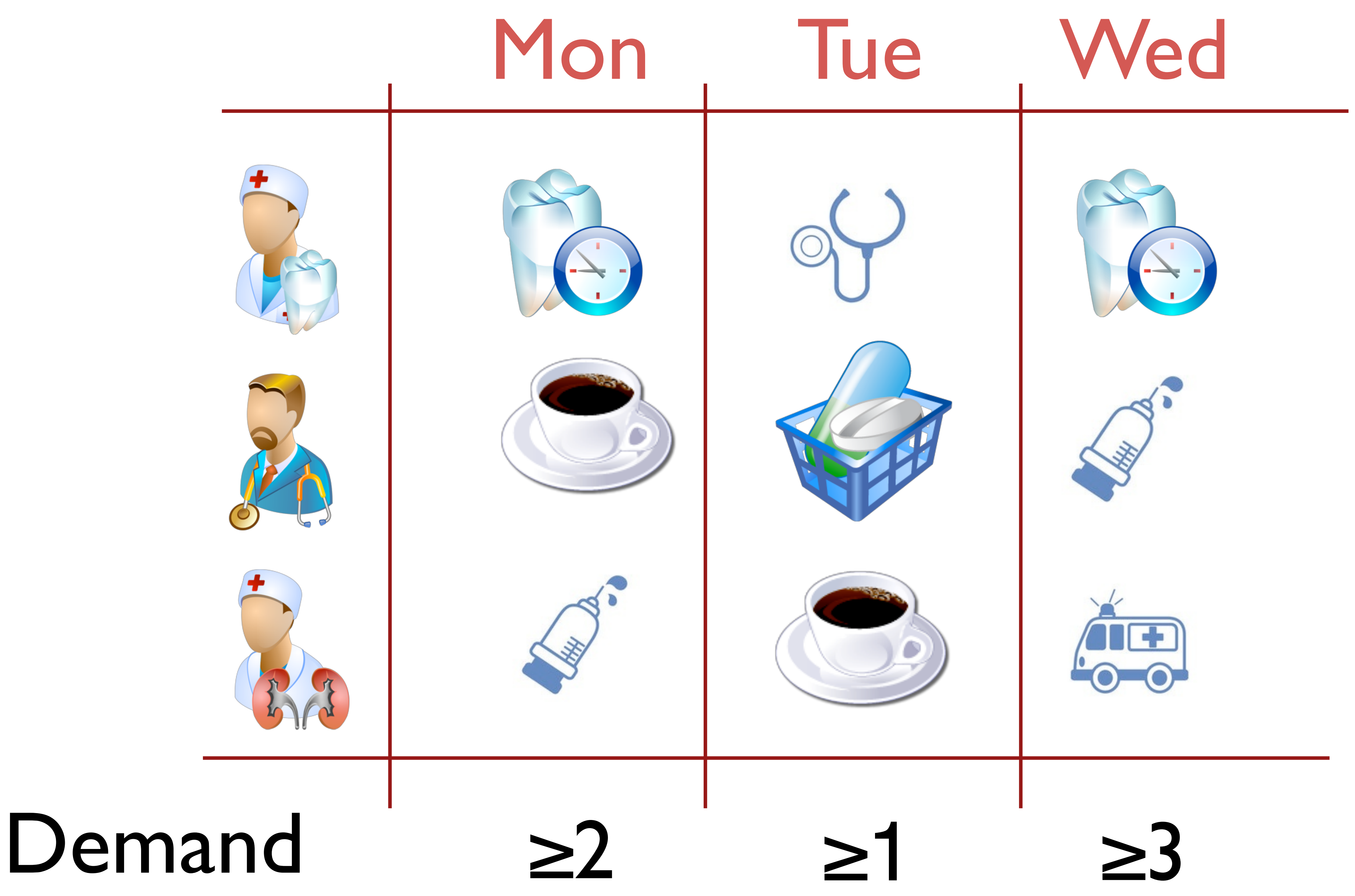
Requests :
None

Fields, Frank

Constraint	Score	Weight	Details
Max consecutive working weekends	0	10000	Requests max 2 consecutive working weekends.
Max working weekends	0	10000	Requests max 2 working weekends.
Max shifts per day	0	10000	Requests max 1 shifts a day.
Max shifts per week	0	10000	Requests max 5 shifts per week.
Max consecutive working days	0	10000	Requests max 5 consecutive working days.
Min consecutive working days	0	10000	Requests min 2 consecutive working days.
Min consecutive shift types	0	10000	Requests min 2 consecutive 'E' shifts Requests min 2 consecutive 'D' shifts Requests min 2 consecutive 'L' shifts Requests min 2 consecutive 'N' shifts

Rostering problems

Nurse Rostering Problem (NRP)



NRP is the problem of finding an optimal way to assign nurses to shifts, typically with a set of hard constraints which all valid solutions must follow, and a set of soft constraints which define the relative quality of valid solutions.
https://en.wikipedia.org/wiki/Nurse_scheduling_problem

Examples of (horizontal) constraints for NRP:

- ✓ A nurse cannot work the day shift, night shift, and late-night shift on the same day (i.e., no 24-hour duties).
- ✓ A nurse may go on a holiday and will not work shifts then.
- ✓ A nurse cannot do a late-night shift followed by a day shift the next day.
- ✓ …

Typically, each such constraint gives rise to a regular expression.

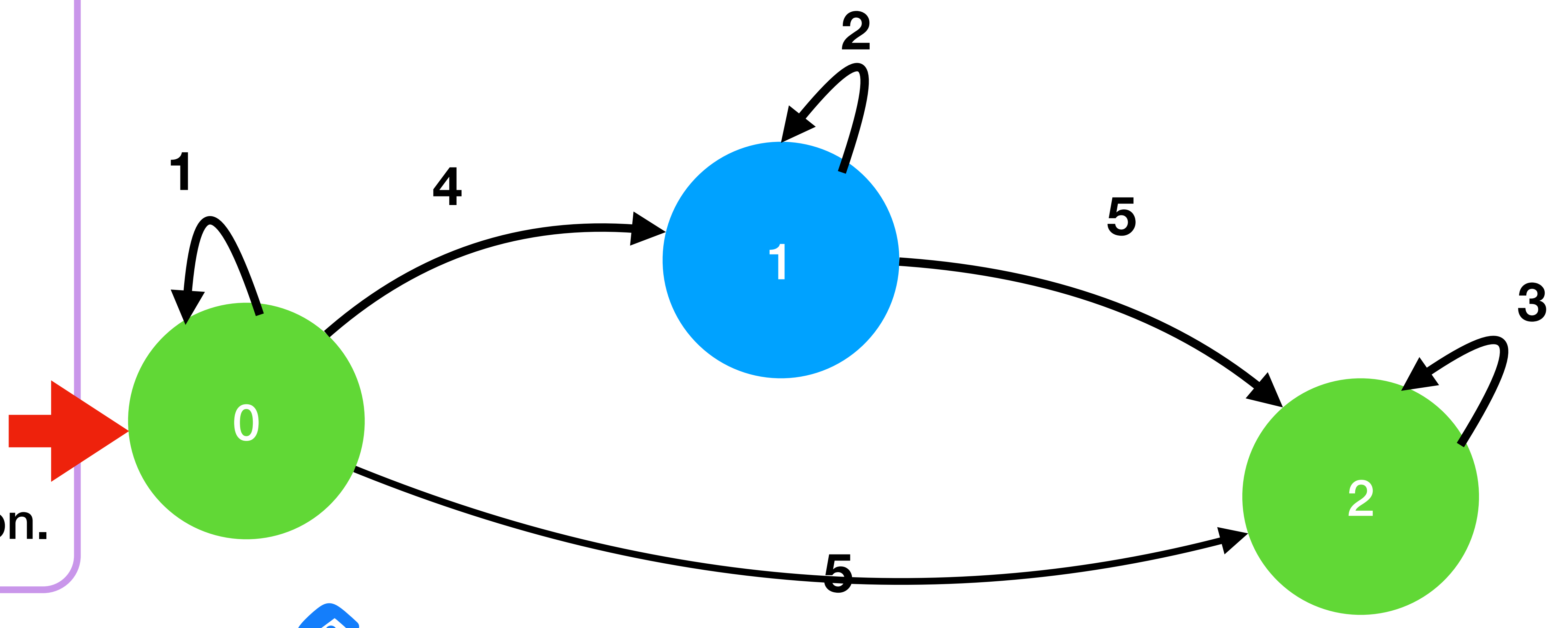
How to enforce rostering constraints?

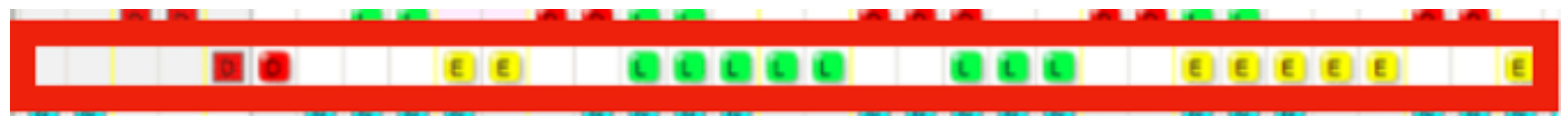
Some examples of constraints are:

✓ A nurse cannot work the day shift, night shift, and late-night shift on the same day (i.e., no 24-hour duties).
✓ A nurse may go on a holiday and will not work shifts then.
✓ A nurse cannot do a late-night shift followed by a day shift the next day.
✓ …

Typically, each such constraint gives rise to a regular expression.

💡By aggregating them into *one* automaton (implementation of regular expression), with transitions and accepting states





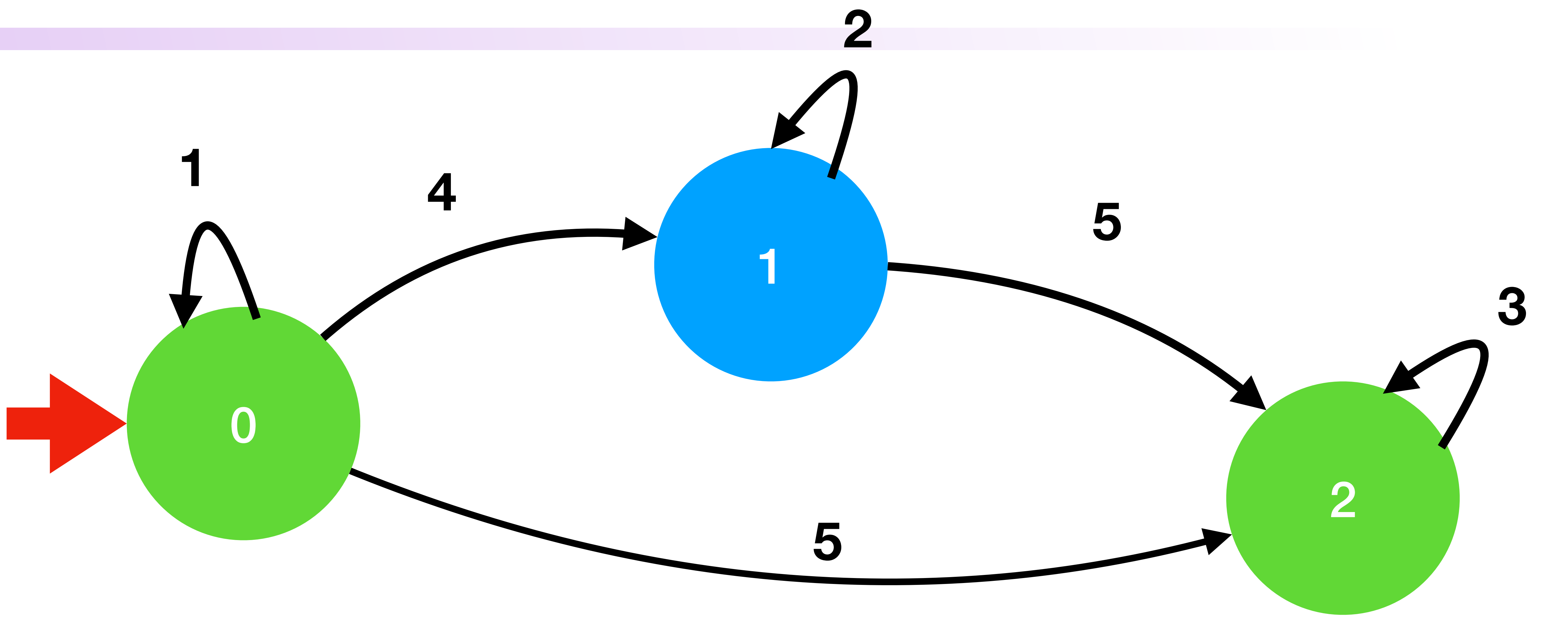
ok iff accepted by automaton

The question is: How do we model in CP an automaton and the acceptance of a string of variables by an automaton?

Regular constraint

(x,y,z) ∊ Language(A)

A:



Constraint: (x,y,z) is a word accepted (at a green state) by the deterministic automaton A, with start state 0.

symbols

states

T	0	1	2	3	4	5
0		0			1	2
1			1			2
2				2		

The model:

Sx = state after reading variable x
Sy = state after reading variables x and y
Sz = state after reading variables x, y, and z
T[s][v] = state after reading at state s the symbol v

Sx = T[0][x]
Sy = T[Sx][y], encode Element2D via Table
Sz = T[Sy][z], encode Element2D via Table
Sz ∊ {0, 2}

Element2D: T[x][y] = z

- Can be modelled with (x,y,z) ∈ table

y

x

	0	1	2	3
0	**1**	**8**	**9**	**6**
1	**1**	**9**	**2**	**4**
2	**9**	**8**	**9**	**8**
3	**1**	**9**	**2**	**5**

- If we have a domain-consistent filtering for Table, then we also have one for Element2D.
- Element2D can be encoded with a Table constraint.

table

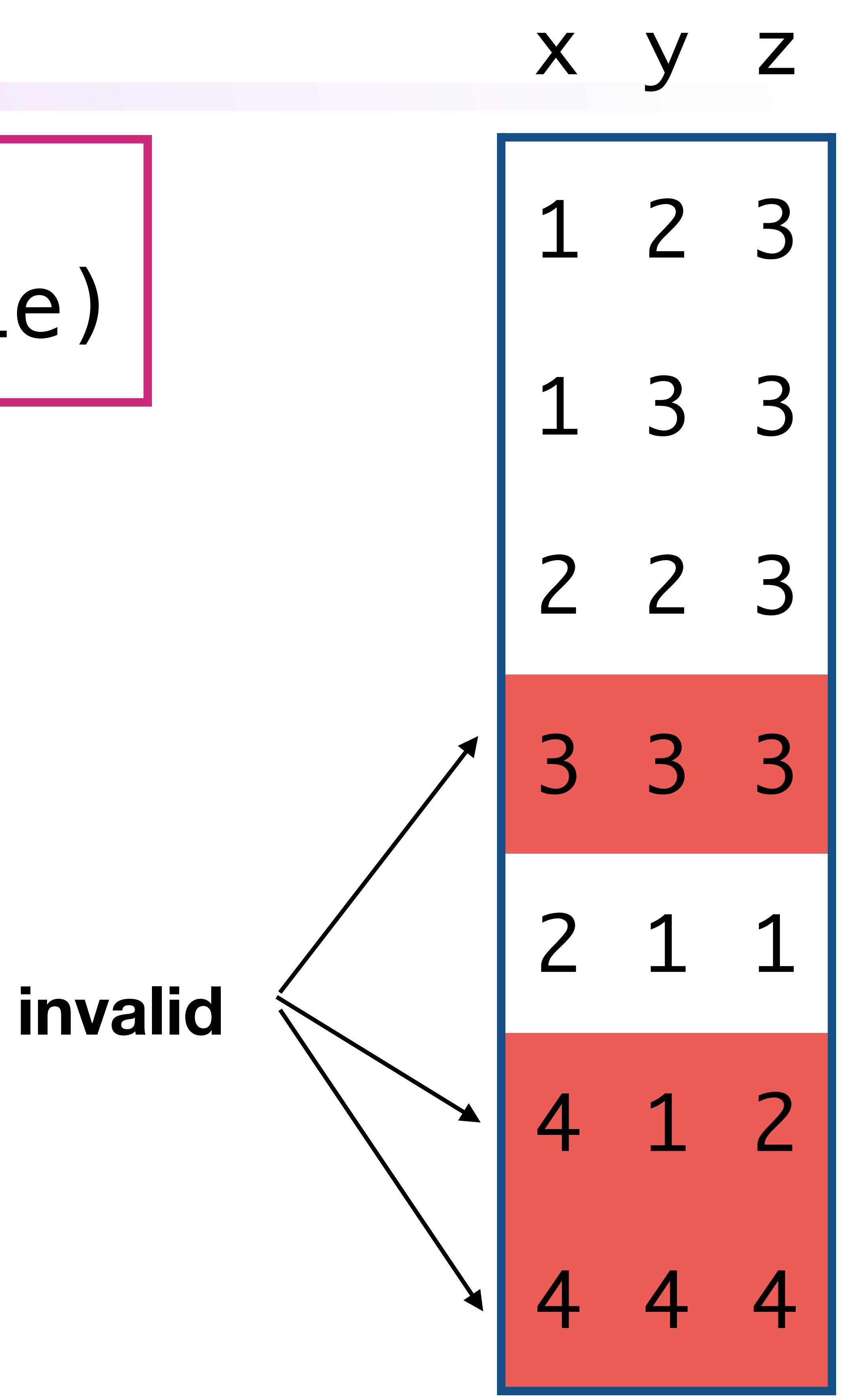
x	y	T[x][y]
0	0	1
0	1	8
0	2	9
0	3	6
1	0	1
1	1	9
1	2	2
1	3	4
2	0	9
2	1	8
2	2	9
2	3	8
3	0	1
3	1	9
3	2	2
3	3	5

Filtering a Table constraint: slow algorithm

Table constraint

```
// x.length = n, dim(table) = m x n
public Table(IntVar[] x, int[][] table)
```

- A tuple (table row) is **valid** iff
  all its values are in the domains of the corresponding variables:
  - valid(table[r]) ≡ ∀i : table[r][i] ∈ D(x[i])
- Literal (x[i],v) is **supported** iff
  there is a valid tuple with value v in column i:
  - ∃r : valid(table[r]) ∧ table[r][i] = v
- Example: D(x) = {1,2}, D(y) = {1,2,3}, D(z) = {1,2,3}
  - (z,3) *is* supported,
    but (z,2) is *not* supported and hence 2 must be removed from D(z).



x	y	z
1	2	3
1	3	3
2	2	3
3	3	3
2	1	1
4	1	2
4	4	4

invalid

A first, slow (but domain-consistent) filtering

```
// x.length=n, dim(table)= m x n
public Table(IntVar[] x, int[][] table)
```

```
SlowTableFiltering(x,table) {
 for ($x_i$ <- x){
    for (v <- $D(x_i)$){
         if ($\nexists r : \forall j \neq i : table(r,j) \in D(x_j) \wedge table(r,i)=v$){
           $D(x_i)$ <- $D(x_i) \setminus \{v\}$
         }
    }
}
```

Slow Table filtering: implementation in MiniCP

```
public void propagate() throws InconsistencyException {
  for (int i = 0; i < x.length; i++) {
    for (int v = x[i].getMin(); v <= x[i].getMax(); v++) {
        if (x[i].contains(v)) {You
          boolean supported = false;
          for (int tupleIdx = 0; tupleIdx < table.length &&
                                 !supported; tupleIdx++) {
              if (table[tupleIdx][i] == v) {
                  boolean allSupported = true;
                  for (int j = 0; j < x.length && allSupported; j++) {
                      if (!x[j].contains(table[tupleIdx][j])) {
                          allSupported = false;
                      }
                     }
                  supported = allSupported;
              }
          }
          if (!supported)
              x[i].remove(v);
      }
    }
  }
}
```

should use your fillArray here

Filtering a Table constraint: the STR algorithm family

Simple Tabular Reduction (STR) algorithms

- Lecoutre, Christophe. STR2: Optimized simple tabular reduction for table constraints. *Constraints,* 2011.

Simple Tabular Reduction (STR) algorithms:

1. For each tuple in the table:
   - The tuple is **valid** ⇒ all its values are in supported literals, so:
     collect the supported literals in a set.
     - Example: (1,2,3) is valid ⇒ (x,1), (y,2), and (z,3) are supported.
   - The tuple is **invalid**:
     remove it (in a stateful way) from the table,
     giving a smaller table, hence incrementality.
2. For each literal (x_i,v):
   if it is not supported (check in the collected set of literals), then remove v from $D(x_i)$.

x	y	z
1	2	3
1	3	3
2	2	3
3	3	3
2	1	1
4	1	2
4	4	4

STR2 algorithm

```
STR2Filtering(x,table) {
   supported = Ø
   for (t <- table) {
      if (∀i: $x_i$.contains(t(i))) {
         for ($x_i$ <- x){
            supported += ($x_i$,t(i))
         }
      } else {
         table.remove(t)
      }
   }
   for ($x_i$ <- x) {
     for (v <- D($x_i$)) {
         if (($x_i$,v)∉supported) {
           D($x_i$) <- D($x_i$) \ {v}
         }
     }
}
```

if tuple is valid

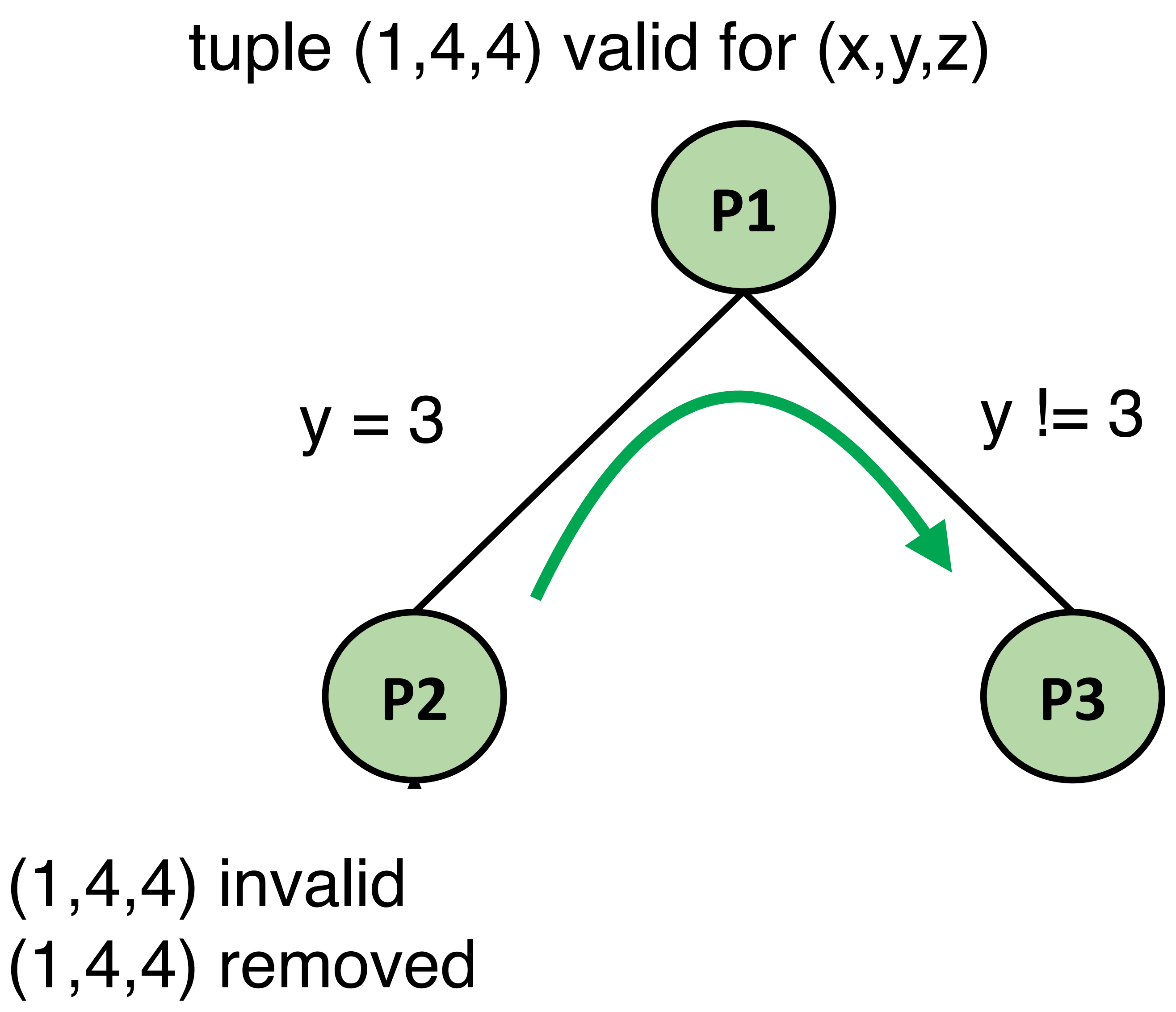
literals collected

else: tuple removed

remove unsupported

Incrementality of STR2

- Incrementality of STR2 comes from the table.
  - Invalid tuples are removed from the table.
  - If a tuple is removed, then it is not inspected in future executions.
- The table has to be stateful (aka reversible), using the state manager, which restores the state on backtrack.



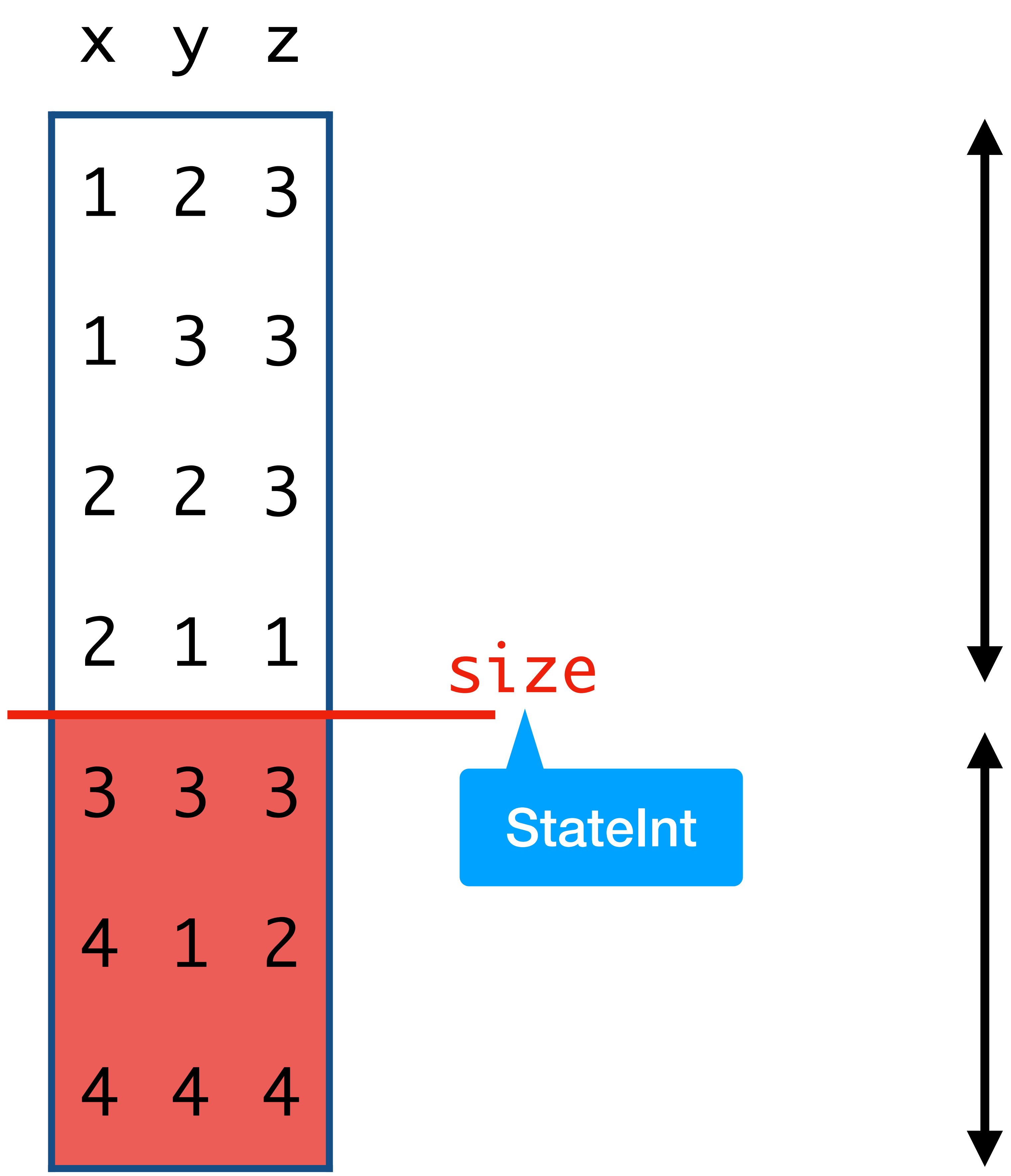
STR2 needs a stateful table

- Use a stateful table (details omitted here) to represent the table.

- All that needs to be backtracked on is just **one integer**, a **StateInt**, denoting the current number of valid tuples in the table, which are stored before the row having that StateInt as index

Stateful table

Assume D(x)={1,2}, D(y)={1,2,3}, D(z)={1,3} now:



x	y	z
1	2	3
1	3	3
2	2	3
2	1	1
3	3	3
4	1	2
4	4	4

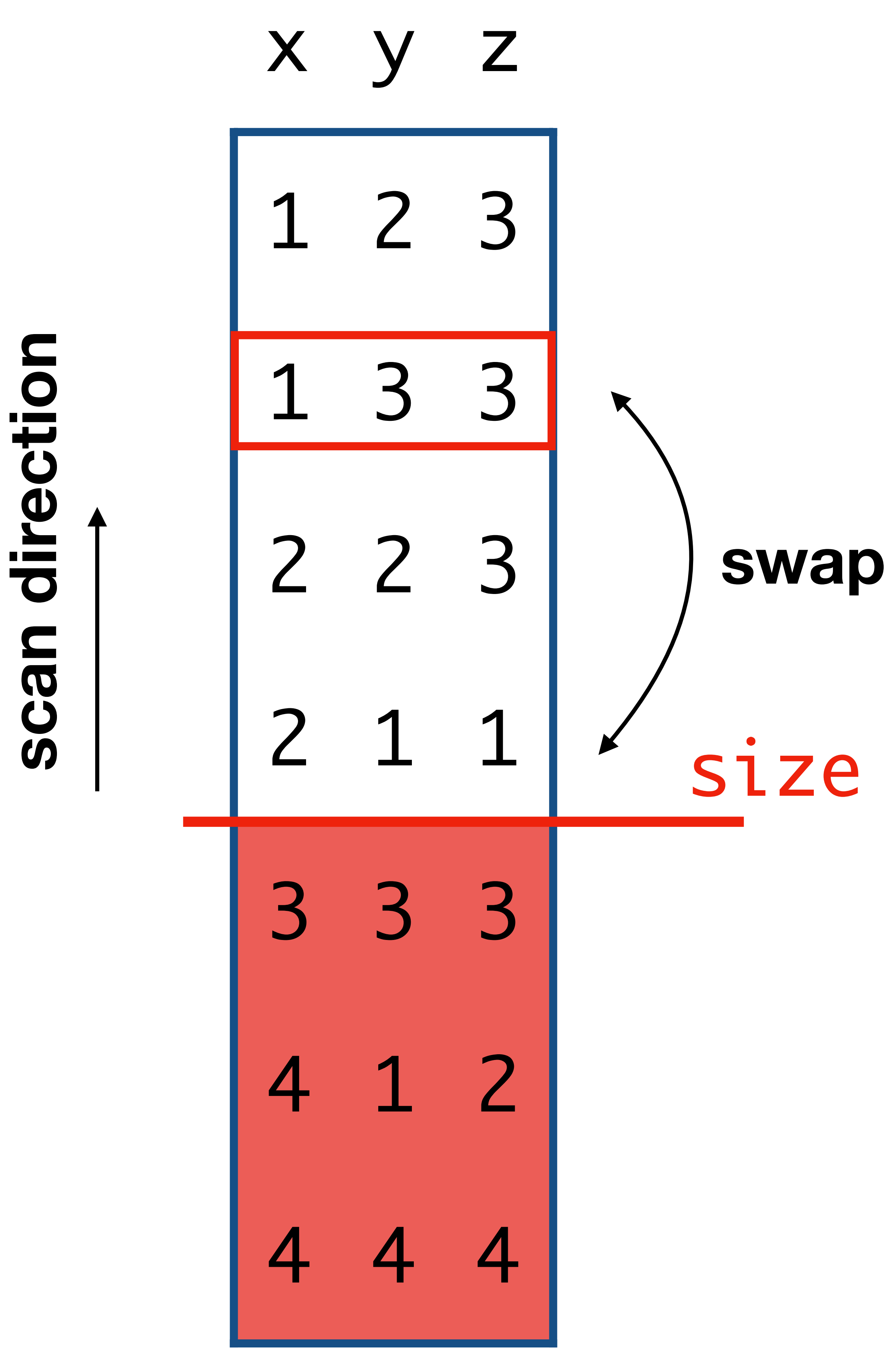
The table is partitioned into two sets:

the remaining tuples are before size

the removed tuples are from size on

Stateful table

Assume 3 is removed from D(y), giving D(x) = {1,2}, D(y) = {1,2}, D(z) = {1,3}:

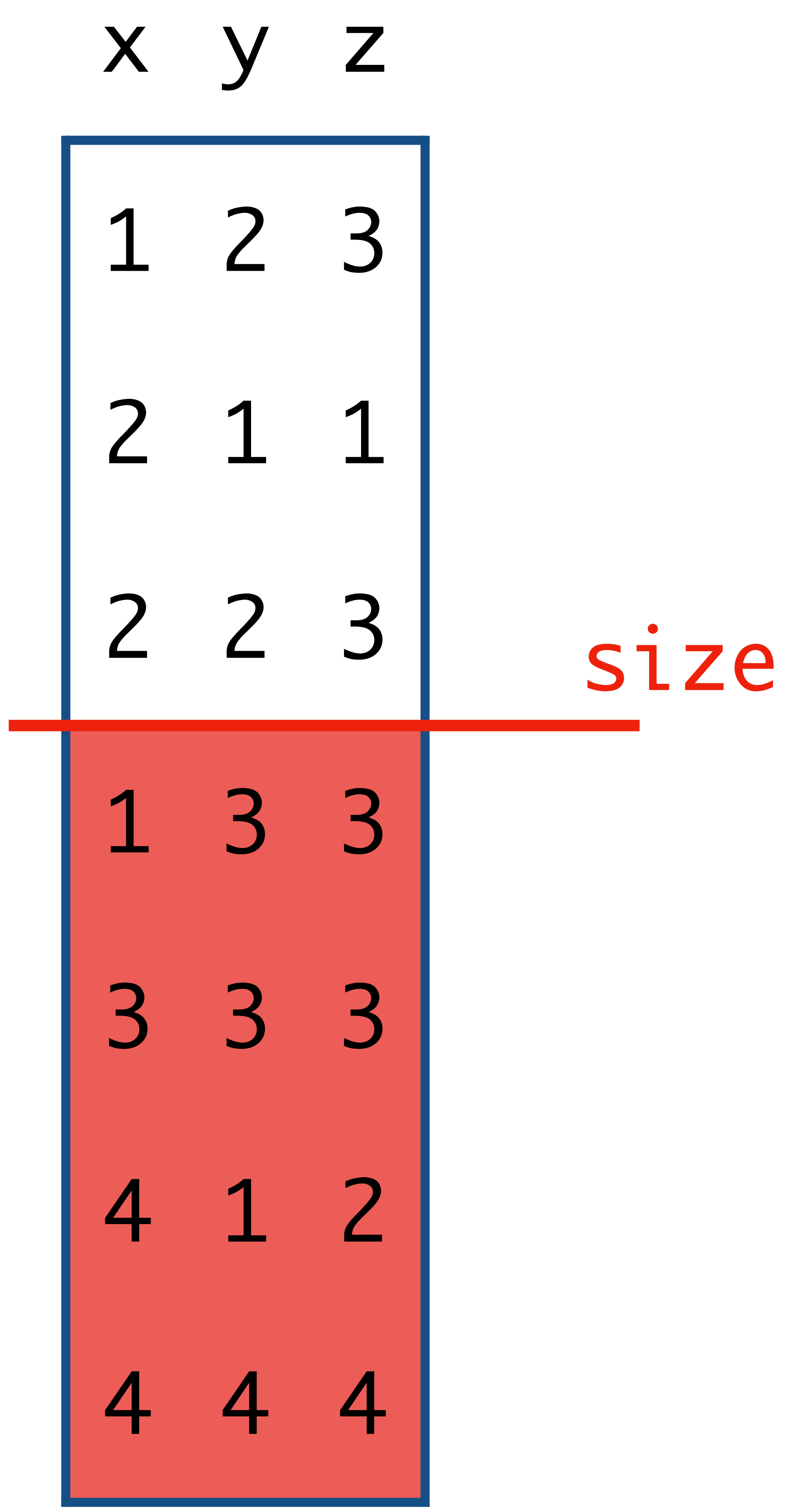


x	y	z
1	2	3
1	3	3
2	2	3
2	1	1
3	3	3
4	1	2
4	4	4

```
When a tuple is removed:
  • it is swapped with the one in position size-1
  • size is decremented
```

Stateful table

Assume 3 is removed from D(y), giving D(x) = {1,2}, D(y) = {1,2}, D(z) = {1,3}:



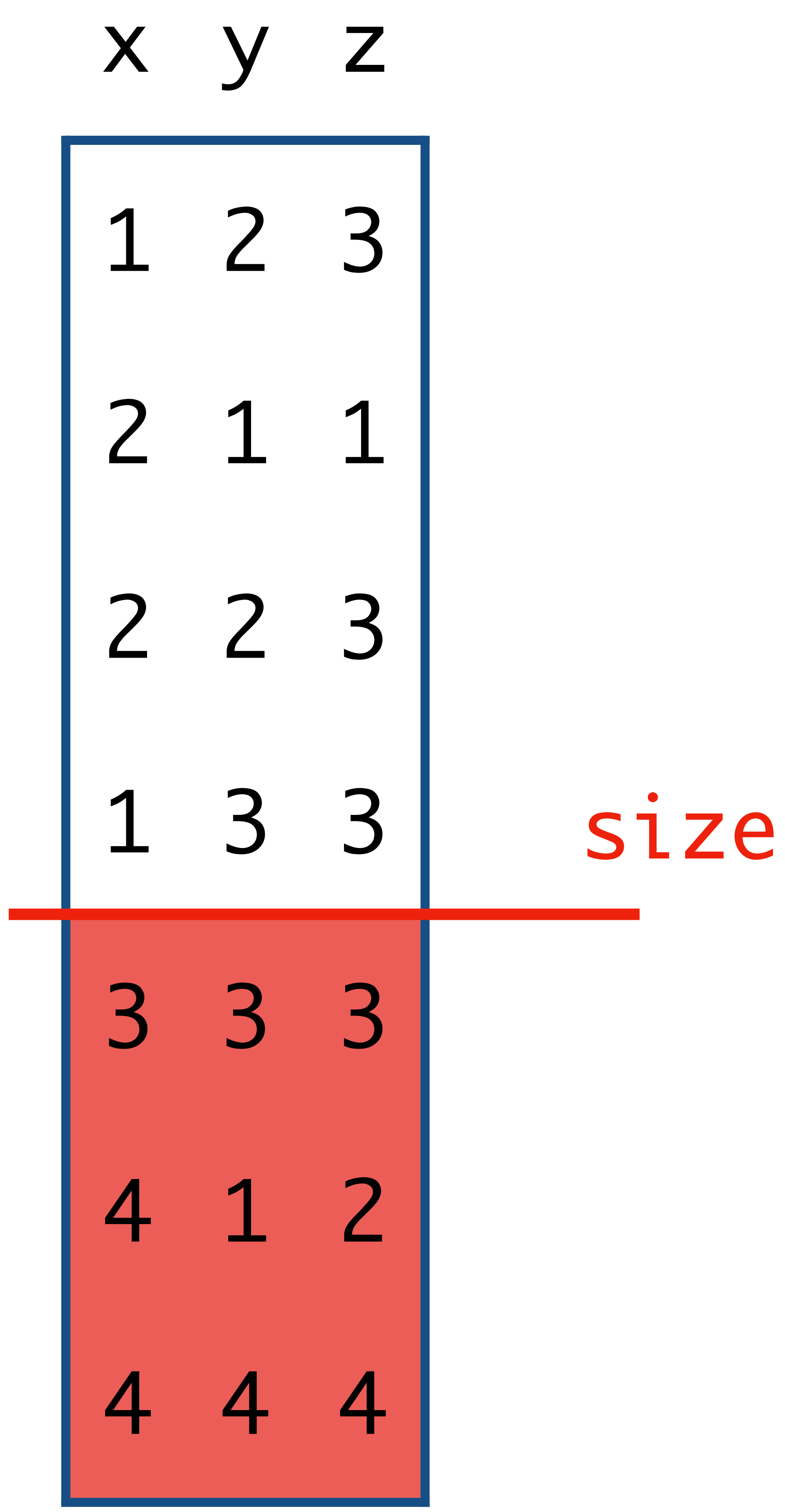
x	y	z
1	2	3
2	1	1
2	2	3
1	3	3
3	3	3
4	1	2
4	4	4

When a tuple is removed:

- it is swapped with the one in position size-1
- size is decremented

Stateful table

Assume restoration to previous state, with D(x) = {1,2}, D(y) = {1,2,3}, D(z) = {1,3}:



x	y	z
1	2	3
2	1	1
2	2	3
1	3	3
3	3	3
4	1	2
4	4	4

On backtracking (sm.restoreState()): restoring size

- restores the removed tuples
- at possibly different positions in the table

Filtering a Table constraint: the Compact Table algorithm

Compact Table: filtering to domain consistency

Demeulenaere, J., Hartert, R., Lecoutre, Ch., Perez, G., Perron, L., Régin, J.-C., & Schaus, P. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. *CP 2016.*

- It is the most efficient known algorithm for filtering a Table constraint to domain consistency.
- It relies on bitwise operations using a data structure called ***reversible/stateful sparse bit set***.
- It is easy to implement (quite similarly to STR2).

Compact Table

index	x	y	z
0	7	5	8
1	2	1	4
2	1	3	2
3	2	4	2
4	6	5	9
5	7	7	8
6	4	2	1
7	1	1	1
8	7	8	9
9	8	9	6
10	2	2	3
11	0	0	0
12	3	3	1
13	5	8	5
14	9	7	7
15	2	3	1

Assume D(x) = D(y) = D(z) = {1,2,3,4,5} in the meantime: what filtering happens now?

Initial list of allowed tuples

Precomputation of support bit sets

$D(x) = D(y) = D(z) = \{1,2,3,4,5\}$

index	x	y	z	supports				
				x=1	x=2	x=3	…	z=5
0	7	5	8	0	0	0		0
1	2	1	4	0	1	0		0
2	1	3	2	1	0	0		0
3	2	4	2	0	1	0		0
4	6	5	9	0	0	0		0
5	7	7	8	0	0	0		0
6	4	2	1	0	0	0		0
7	1	1	1	1	0	0		0
8	7	8	9	0	0	0		0
9	8	9	6	0	0	0		0
10	2	2	3	0	1	0		0
11	0	0	0	0	0	0		0
12	3	3	1	0	0	1		0
13	5	8	5	0	0	0		1
14	9	7	7	0	0	0		0
15	2	3	1	0	1	0		0

Bit Sets

Every bit supports(x_i,v)(r) is computed when posting the constraint:

- 1 if table[r][i]=v
- 0 otherwise

Can we identify the valid tuples (green ones) from the supports(x_i,v) bit sets?

Computation of each validTuples(r)

index	validTuples	x	y	z	supports				
					x=1	x=2	x=3	…	z=5
0	0	7	5	8	0	0	0		0
1	1	2	1	4	0	1	0		0
2	1	1	3	2	1	0	0		0
3	1	2	4	2	0	1	0		0
4	0	6	5	9	0	0	0		0
5	0	7	7	8	0	0	0		0
6	1	4	2	1	0	0	0		0
7	1	1	1	1	1	0	0		0
8	0	7	8	9	0	0	0		0
9	0	8	9	6	0	0	0		0
10	1	2	2	3	0	1	0		0
11	0	0	0	0	0	0	0		0
12	1	3	3	1	0	0	1		0
13	0	5	8	5	0	0	0		1
14	0	9	7	7	0	0	0		0
15	1	2	3	1	0	1	0		0

If row r is supported, then 1, else 0

validTuples =
(supports(x,1) | supports(x,2) | supports(x,3) | supports(x,4) | supports(x,5)) **&**
(supports(y,1) | supports(y,2) | supports(y,3) | supports(y,4) | supports(y,5)) **&**
(supports(z,1) | supports(z,2) | supports(z,3) | supports(z,4) | supports(z,5))

Compact Table: domain-consistency filtering

Goal: remove values not supported anymore

```
CompactTableFiltering(x,table) {
 for ($x_i$ <- x) {
    for (v <- D($x_i$)) {
       if (validTuples & supports($x_i$,v) = 0) {
           D($x_i$) <- D($x_i$) \ {v}
        }
    }
 }
}
```

`validTuples & supports(`x_i`,v) = 0`
(where `0` is the all-zero bit set) is implemented
in the Java class `BitSet` using method `intersects`

Compact Table: filtering example

index	validTuples	x	y	z	supports		
					x=5	y=5	z=5
0	0	7	5	8	0	1	0
1	1	2	1	4	0	0	0
2	1	1	3	2	0	0	0
3	1	2	4	2	0	0	0
4	0	6	5	9	0	1	0
5	0	7	7	8	0	0	0
6	1	4	2	1	0	0	0
7	1	1	1	1	0	0	0
8	0	7	8	9	0	0	0
9	0	8	9	6	0	0	0
10	1	2	2	3	0	0	0
11	0	0	0	0	0	0	0
12	1	3	3	1	0	0	0
13	0	5	8	5	1	0	1
14	0	9	7	7	0	0	0
15	1	2	3	1	0	0	0

```
validTuples & supports(x,5) = 0
validTuples & supports(y,5) = 0
validTuples & supports(z,5) = 0
```

D(x)={1,2,3,4,~~5~~}

D(y)={1,2,3,4,~~5~~}

D(z)={1,2,3,4,~~5~~}

Update of validTuples when a domain change occurs

- Assume 1 and 2 are now removed from D(x) = {~~1,2~~,3,4}.
- We first need to update **validTuples**. There are two possible strategies:

1. **From scratch**, based on the remaining values of *all* variable domains: D(x) = {3,4} and D(y) = D(z) = {1,2,3,4}. Same as the initial computation:

   validTuples = (supports(x,3) | supports(x,4)) & (supports(y,1) | supports(y,2) | supports(y,3) | supports(y,4)) & (supports(z,1) | supports(z,2) | supports(z,3) | supports(z,4))

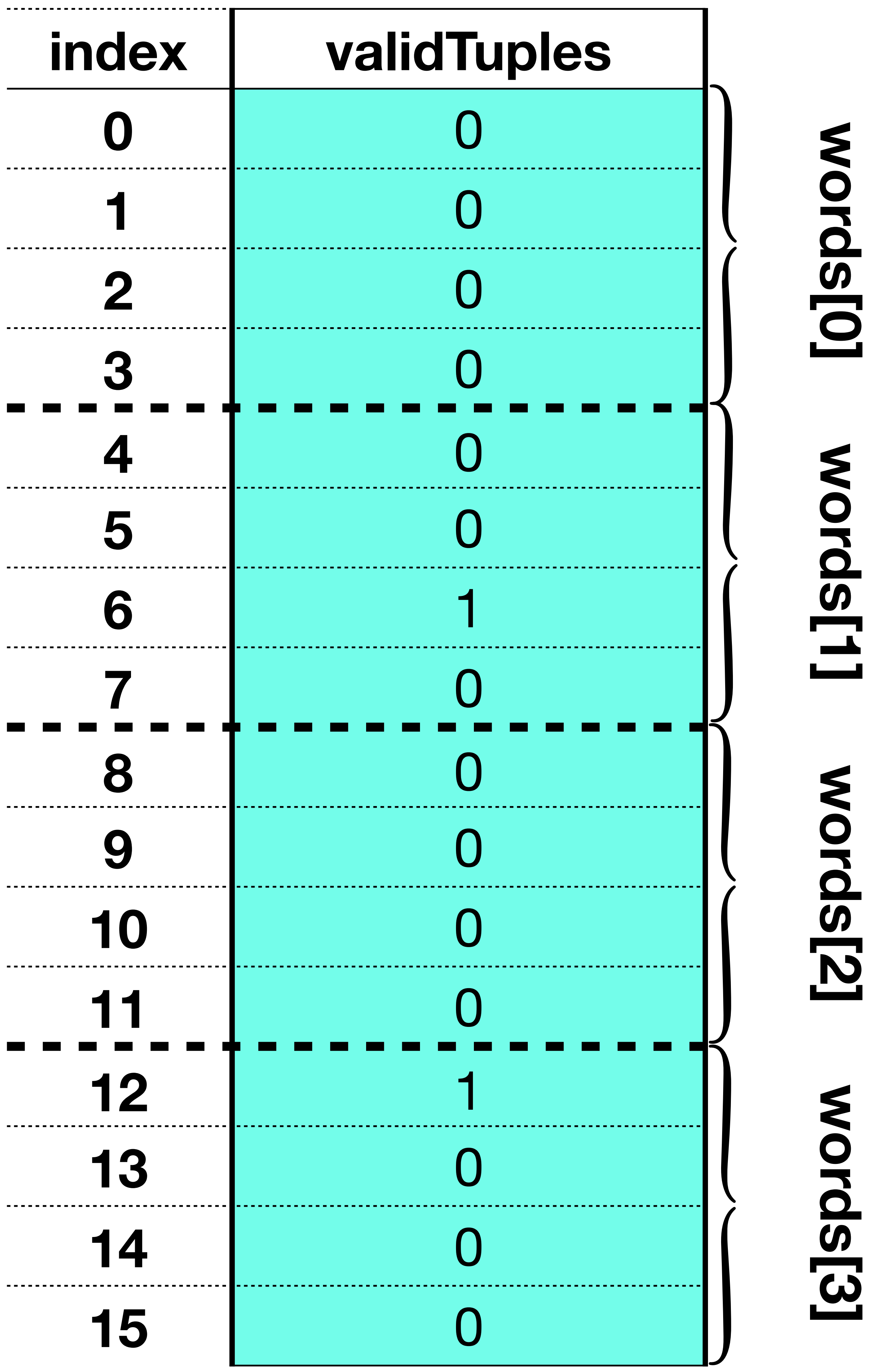
2. **Incrementally**, based on *the* modified variable domain: D(x) = {3,4}.

   validTuples = **validTuples** & (supports(x,3) | supports(x,4))

   Indeed, dropping the influence of bit set *a* on the bit set (*a* | *b*) & *c*, so as to get *b* & *c*, can also be done by computing ((*a* | *b*) & *c*) & *b*.

Underlying data structure: the StateSparseBitSet API

Compact Table and state restoration



index	validTuples
0	0
1	0
2	0
3	0
4	0
5	0
6	1
7	0
8	0
9	0
10	0
11	0
12	1
13	0
14	0
15	0

In practice, assume Long of 64 bits ;-)

- The update requires having a stateful validTuples bit set: it must recover on backtrack (sm.restoreState, …).
- We introduce a data structure called StateBitSet that encapsulates an array of StateLong (of 64 bits each).
- This data structure represents validTuples:

validTuples: StateBitSet

words = StateLong[ ]

Can we further improve the efficiency?

Yes, as bitwise operations do not need to be computed on words that are zero!

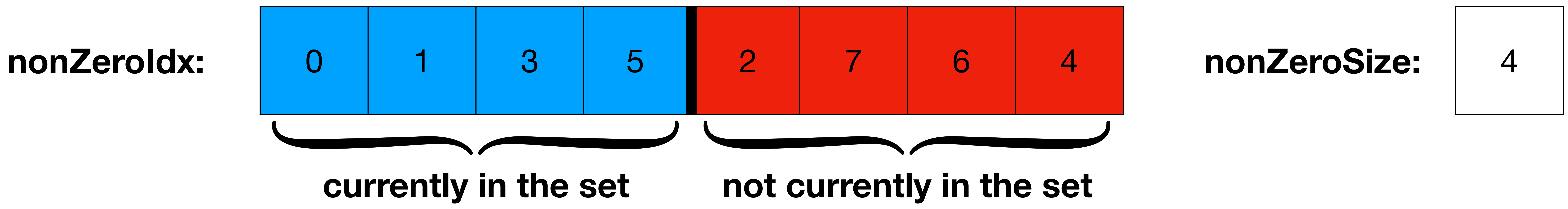
	Index	validTuples	x	y	z	supports(y,1)
words[0]	0	0	7	5	8	0
	1	0	2	1	4	1
	2	0	1	3	2	0
	3	0	2	4	2	0
words[1]	4	0	6	5	9	0
	5	0	7	7	8	0
	6	1	4	2	1	0
	7	0	1	1	1	1
words[2]	8	0	7	8	9	0
	9	0	8	9	6	0
	10	0	2	2	3	0
	11	0	0	0	0	0
words[3]	12	1	3	3	1	0
	13	0	5	8	5	0
	14	0	9	7	7	0
	15	0	2	3	1	0

$D(x) = \{3,4\}, \ \ D(y) = \{1,2,3,4\} = D(z)$

```
CompactTableFiltering(x,table) {
  for (x_i <- x) {
    for (v <- D(x_i)) {
       if (validTuples & supports(x_i,v) = 0) {
          D(x_i) <- D(x_i) \ {v}
       }
    }
  }
}
```

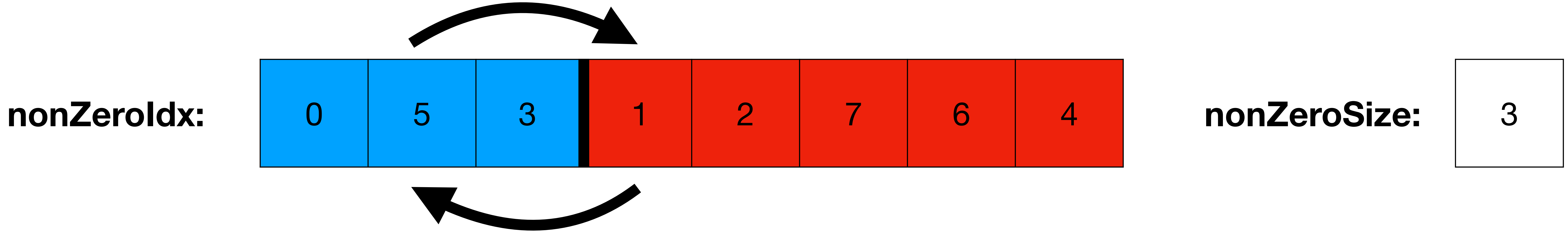
How can we do that efficiently?

- Use an internal sparse set for the state
- Goal: maintain the set of indices of non-zero words



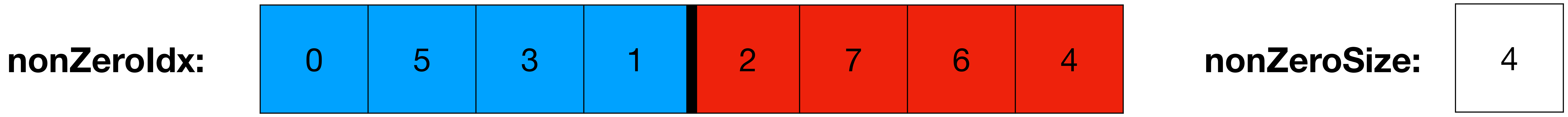
nonZeroIdx:	0	1	3	5	2	7	6	4	**nonZeroSize:**	4

- If a word becomes zero, then it is swapped with the last non-zero word



nonZeroIdx:	0	5	3	1	2	7	6	4	**nonZeroSize:**	3

- Restoration requires only the size variable to be restored



nonZeroIdx:	0	5	3	1	2	7	6	4	**nonZeroSize:**	4

Can we even further improve the efficiency?

- Yes, as the last intersecting word is more likely to intersect again
- Remember the index, called ***residue***, of the last word that led to a non-empty intersection and try it first for the next intersection test with some supports(x_i,v)

```
boolean intersects(validTuples,supports[x,v]) {
  residue = support[x,v].residue
  if (validTuples.words[residue] & supports[x,v].words[residue] != 0L)
    return true
  i = nonZeroSize
  while (i > 0){
    i = i - 1
    wordIdx = nonZeroIdx[i]
    if (validTuples.words[wordIdx] & supports[x,v].words[wordIdx] != 0L){
      supports[x,v].residue = wordIdx
      return true
    }
  }
  return false
}
```

residue = int value (cache) that remembers the last word proving non-empty intersection: O(1) check instead of O(|words|)

new last-known intersecting word, hence new residue

How can we do that efficiently?

All this can be implemented in a data structure called StateSparseBitSet, with the following API:

```
mask.clear() // empty the bit set
mask.or(supports[x,a]) // mask = mask | supports[x,a]
validTuples.and(mask) // words = words & mask
validTuples.intersects(supports[x,c]) // words & supports[x,c] != 0L
```

Update validTuples with StateSparseBitSet API

validTuples = validTuples & (supports(x,3) | supports(x,4))

```
1. mask.clear()
2. mask.or(supports[x,3])
3. mask.or(supports[x,4])
4. validTuples.and(mask)
```

Filtering validTuples with StateSparseBitSet API

Is x=3 still possible?

```
answer = validTuples.intersects(supports[x,3])
if (!answer) { x.remove(3)}
```

StateSparseBitSet

- **Set**: collection of objects
- **BitSet**: uses bits (one dedicated bit per object) to represent the presence (bit set to 1) or absence (bit set to 0) of an object in the set
- **Sparse**: optimized to avoid computations on empty parts of the data structure
- **State**: allow automatic restoration to a previous saved state