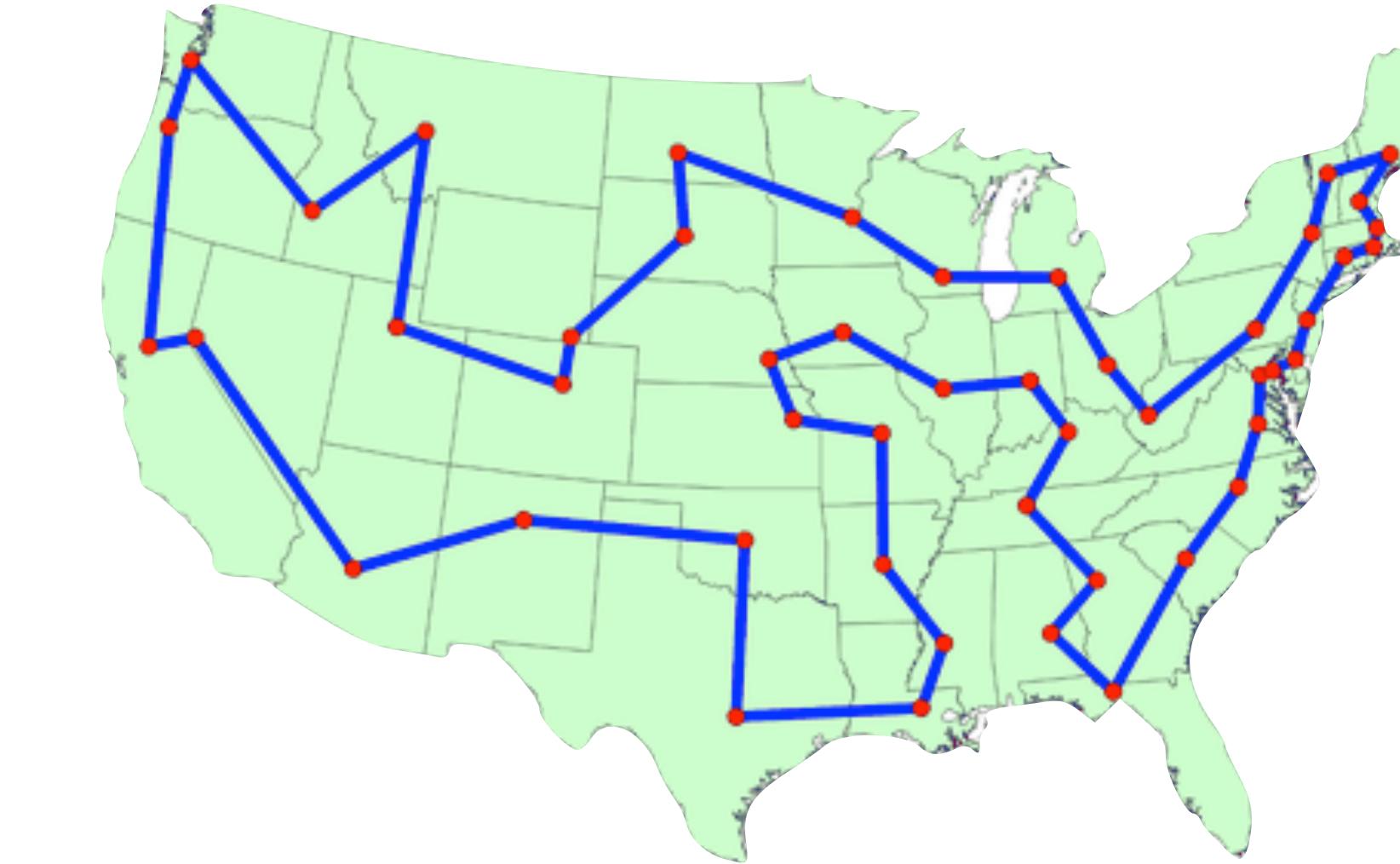




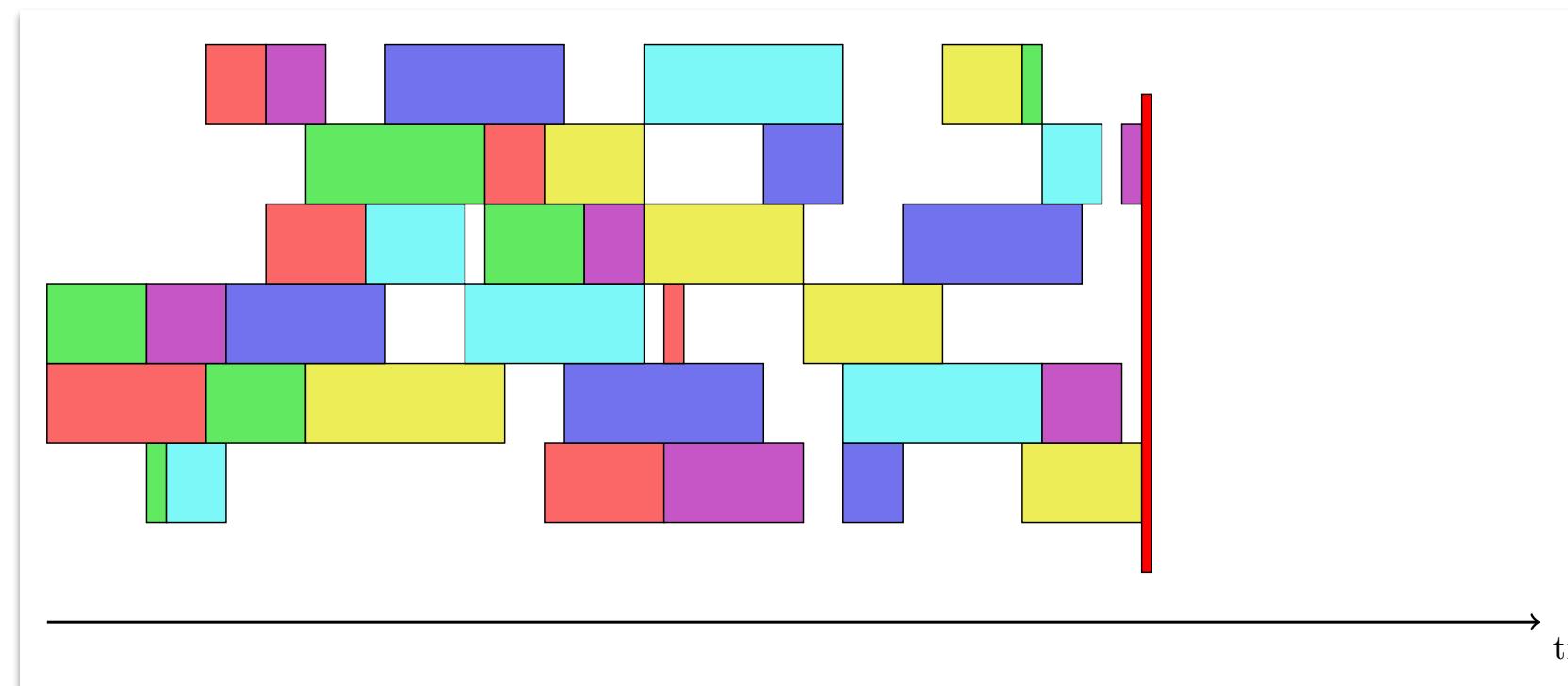
# Introduction to Constraint Programming

# Discrete Optimization is everywhere!

## Routing



## Scheduling

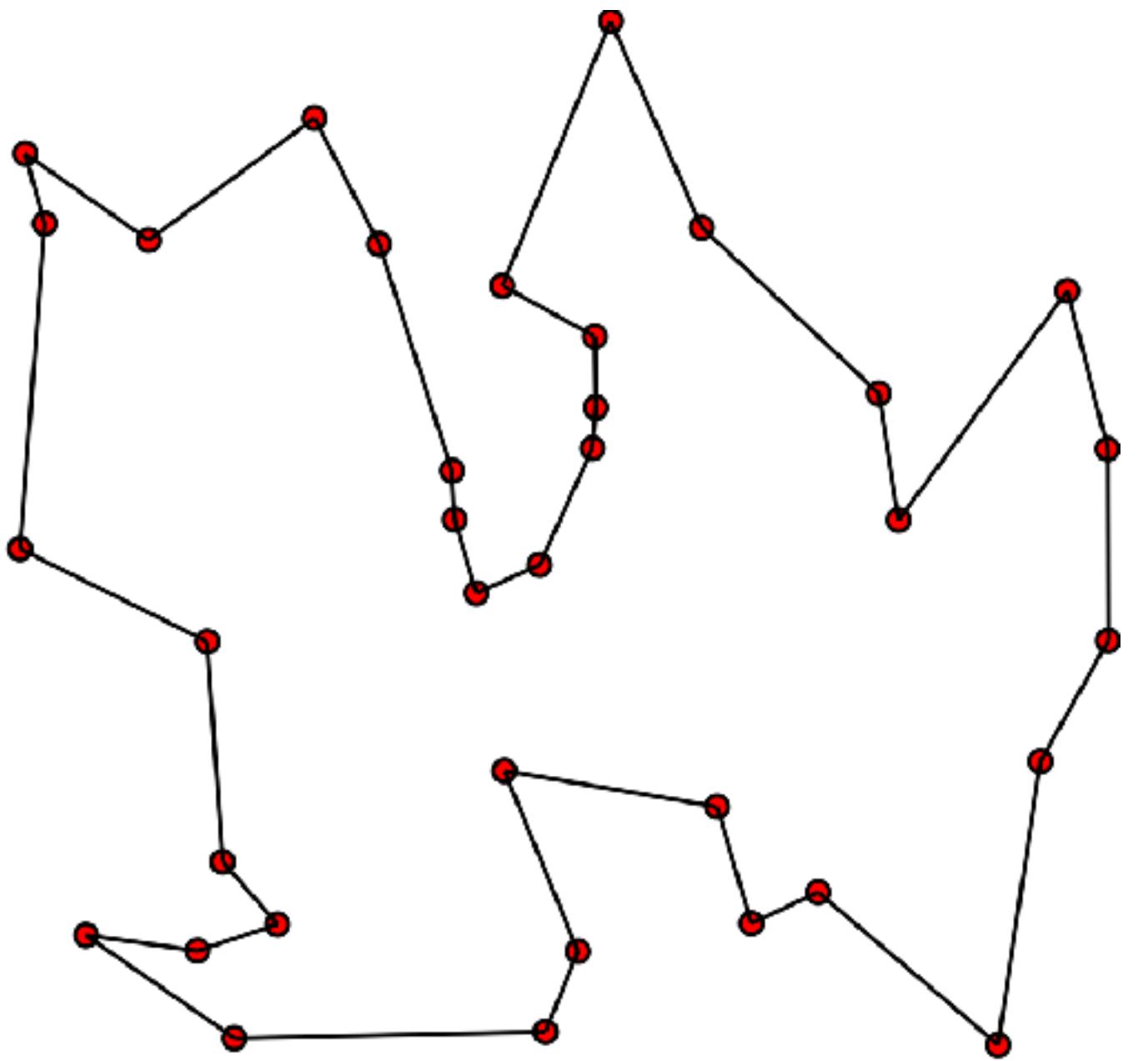


## Rostering

Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon							
6	14	22	6	14	22	6	14	22	6	14	22	6	14	22
Maximum consecutive working days for Ann: 5														
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
A	?	?	A	?	?	A	?	?	A	?	?	A	?	?
1	2	3	4	5	6	7								
Minimum consecutive free days for Beth: 2														
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
?	B	?	?	?	?	B	?	?	?	?	?	?	C	?
1	2												F	
After a night shift sequence: 2 free days      Unwanted pattern: E-L-E														
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
?	?	D	?	?	D	?	?	?	D	?	?	E	?	E
N	N				F				E		L		E	

# Discrete Optimization problems are messy

- Pure TSP only exist in text-books and student projects



- In practice you will have more than one vehicle, and dozens of constraints and strange objective functions 😜

# Constraint Programming

- Is a very good tool to solve messy discrete optimization problems



# Constraint Programming (CP)

“Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.” (E. Freuder)

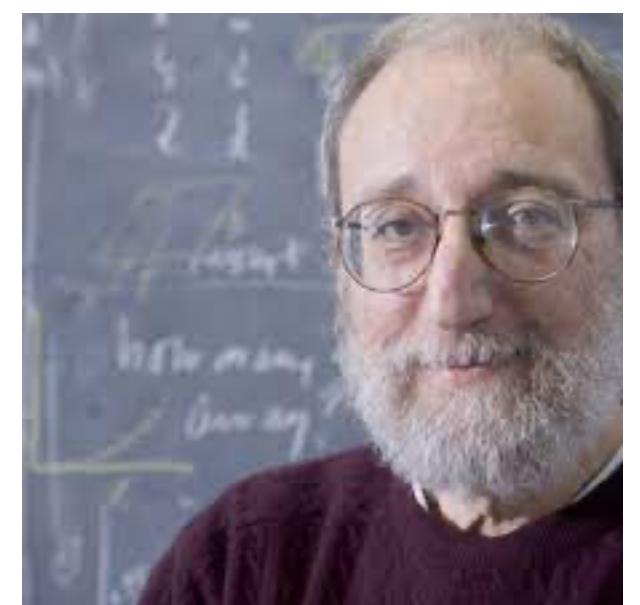


States, you mean like this?

Not yet ... rather like this:

```
range R = 1..8;  
var{int} q[R] in R;  
solve {  
    forall(i in R, j in R: i < j) {  
        q[i] ≠ q[j];  
        q[i] ≠ q[j] + (j - i);  
        q[i] ≠ q[j] - (j - i);  
    }  
}
```

but who knows in the future ;-)



# State Problem = Declarative Programming

Declarative programming is a *programming paradigm* that expresses the logic of a computation without describing its control flow.

Declarative programming for solving constrained combinatorial (optimization) problems means that you express the properties of solutions that must be found by “the solver”.

# CP Slogan

## CP = Model (+ Search)

Model description:  
user API for  
declarative programming

The algorithmic part:  
finding a solution that  
satisfies all the constraints, etc,  
usually by exploring a search tree



# What will you learn ?

- 1) How to build this
- 2) How to use this



# Outline

---

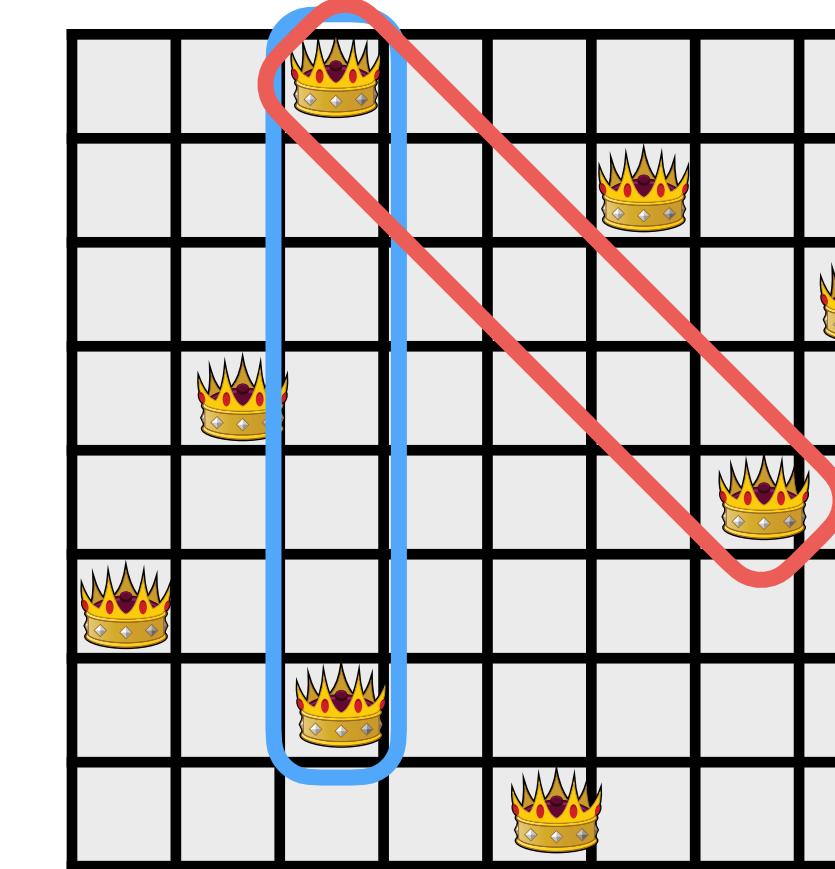
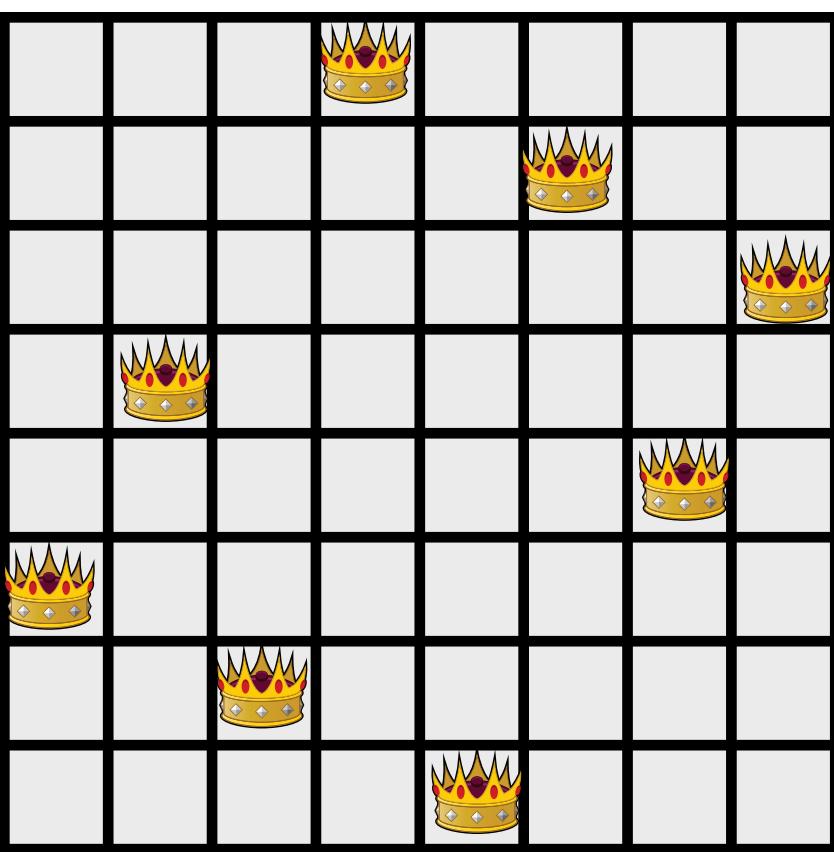
- ▶ The N-Queens Problem
- ▶ Three approaches
  - DFS + Filter
  - DFS + Prune
  - (Tiny)-CSP: make it generic and reusable:
    - Variables, domains, constraints and DFS
  - Declarative Paradigm
  - Assignment: MagicSquare + Killer Sudoku
  - What's next



# DFS + Filter

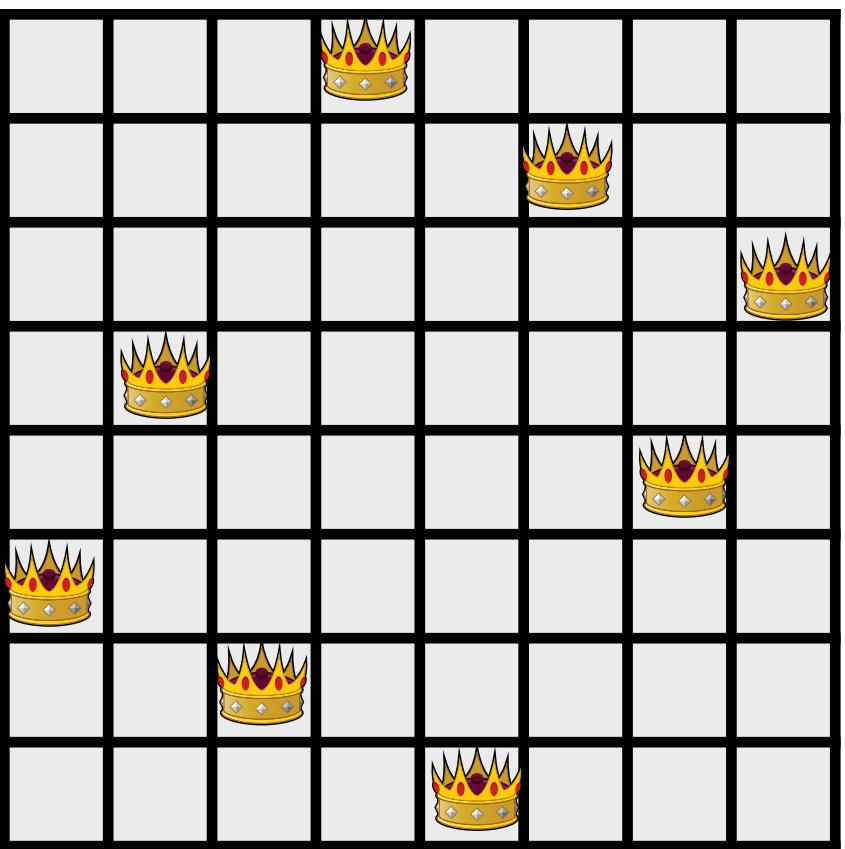
# N-Queens Problem

- ▶ Place eight queens on an  $n \times n$  chessboard so that no two queens threaten each other;
- ▶ Thus, a solution requires that no two queens share the same row, column, or diagonal.



# N-Queens: modeling considerations

A boolean {True/False} for each cell telling whether or not a queen is present



F	F	F	T	F	F	F	F
F	F	F	F	F	T	F	F
F	F	F	F	F	F	F	T
F	T	F	F	F	F	F	F
F	F	F	F	F	F	T	F
T	F	F	F	F	F	F	F
F	F	T	F	F	F	F	F
F	F	F	F	T	F	F	F

# N-Queens: modeling considerations

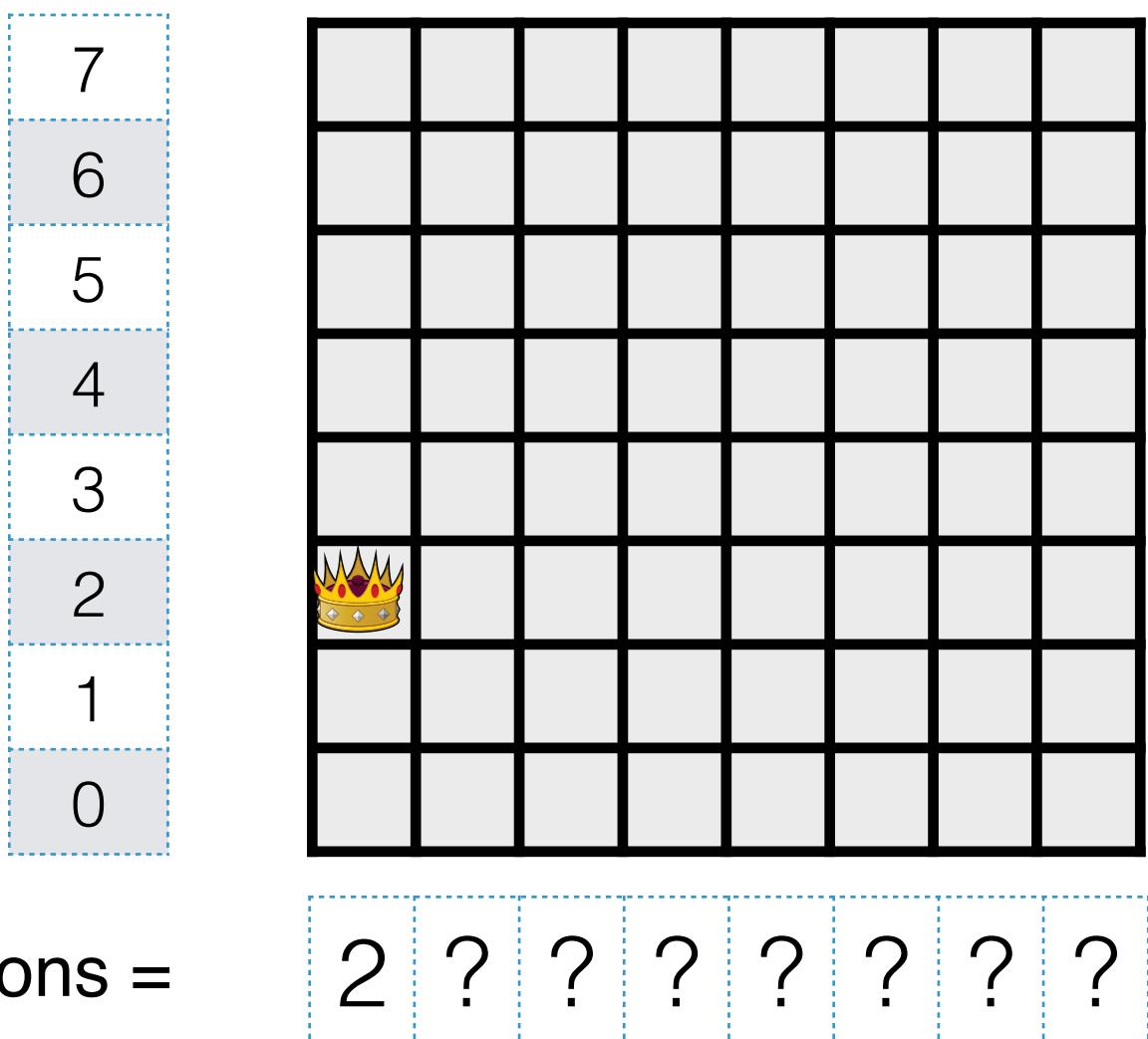
A boolean {True/False} for each cell telling whether or not a queen is present

Drawback: Require to test the three types of constraints: no two queens share the **same row**, **column**, or **diagonal**.

F	F	F	T	F	F	F	F
F	F	F	F	F	T	F	F
F	F	F	F	F	F	F	T
F	T	F	F	F	F	F	F
F	F	F	F	F	F	T	F
T	F	F	F	F	F	F	F
F	F	T	F	F	F	F	F
F	F	F	F	T	F	F	F

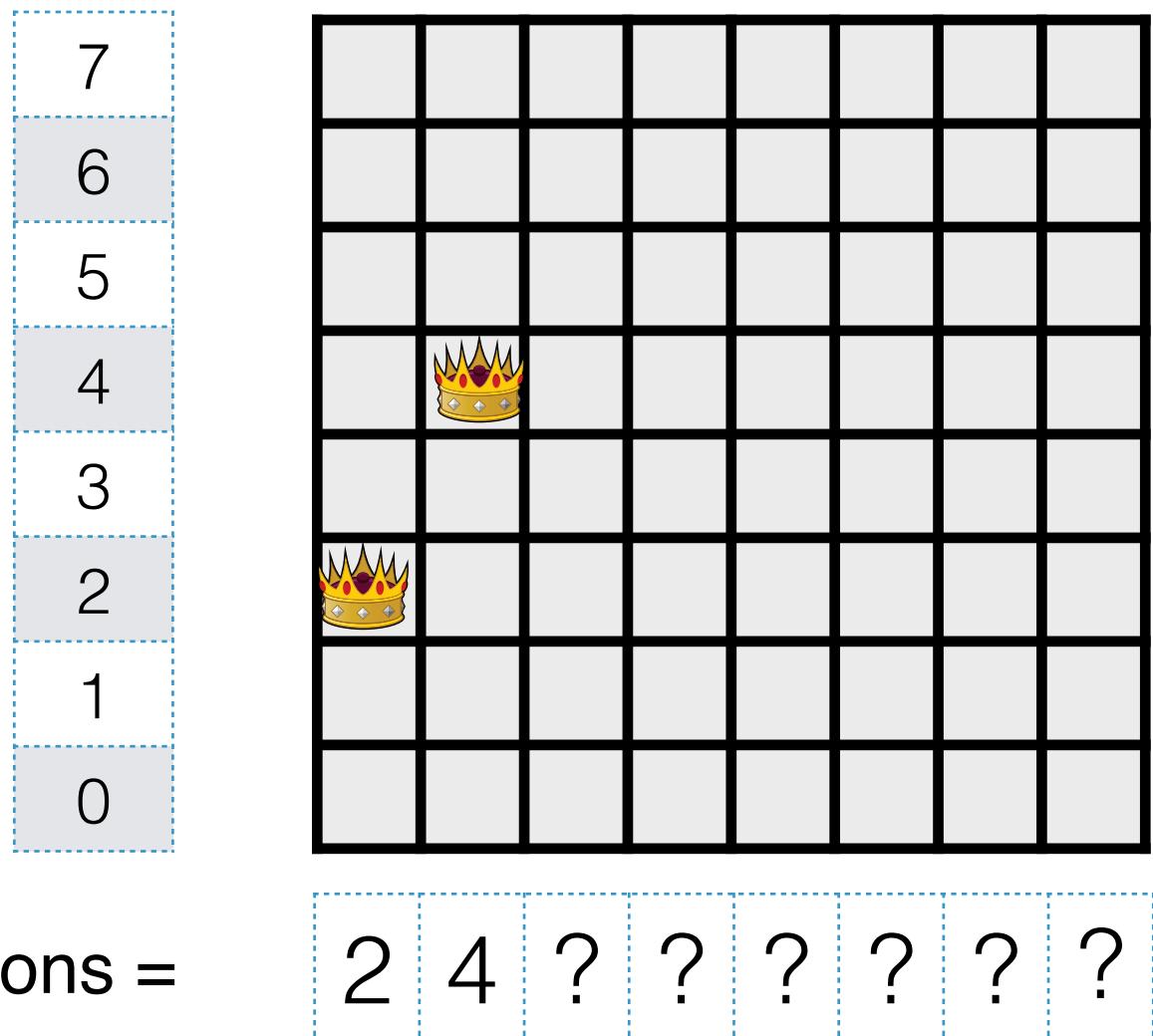
# N-Queens: modeling considerations

An integer for each column  $\{0, \dots, N-1\}$  telling in which row to place the queen



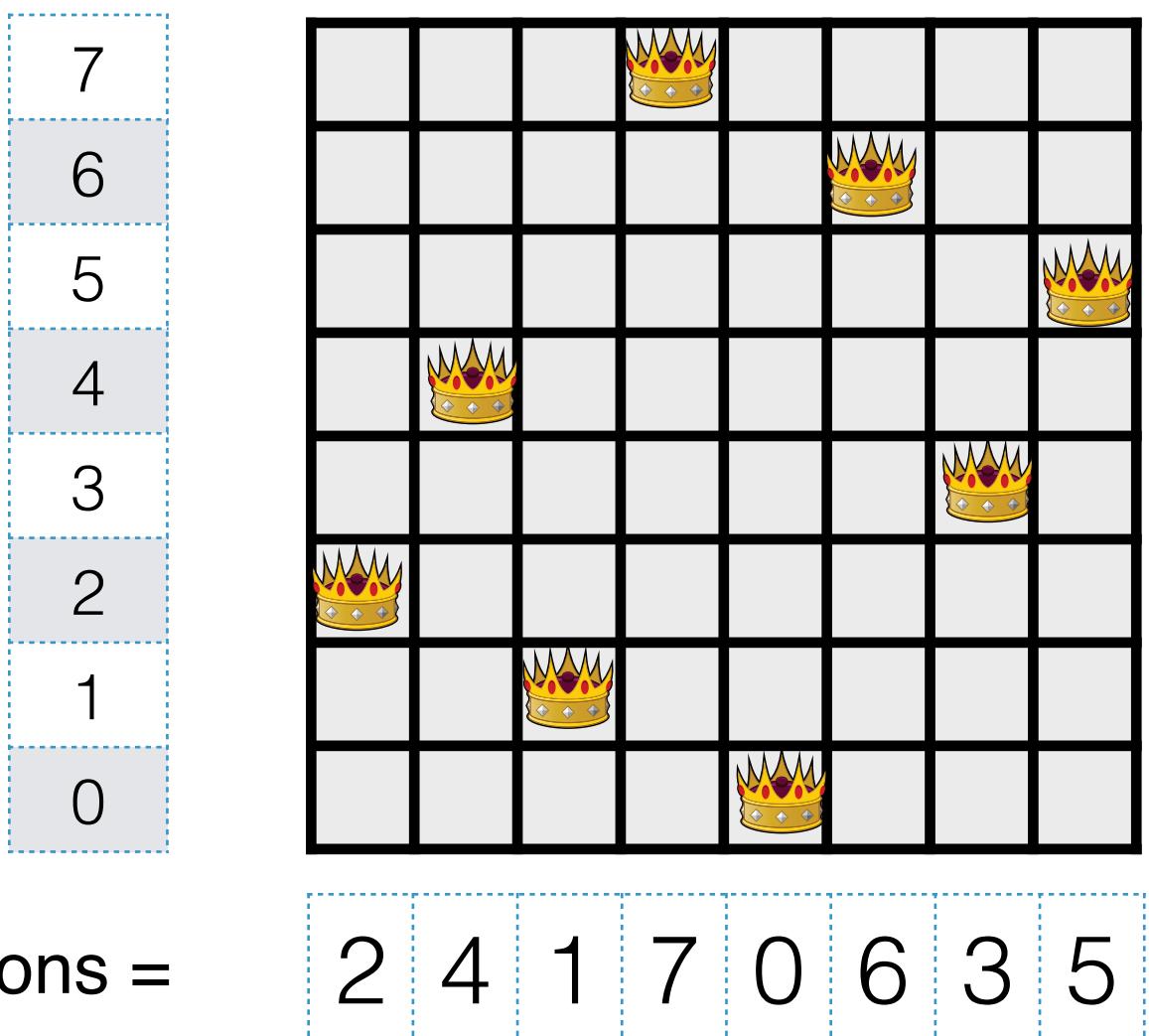
# N-Queens: modeling considerations

An integer for each column  $\{0, \dots, N-1\}$  telling in which row to place the queen



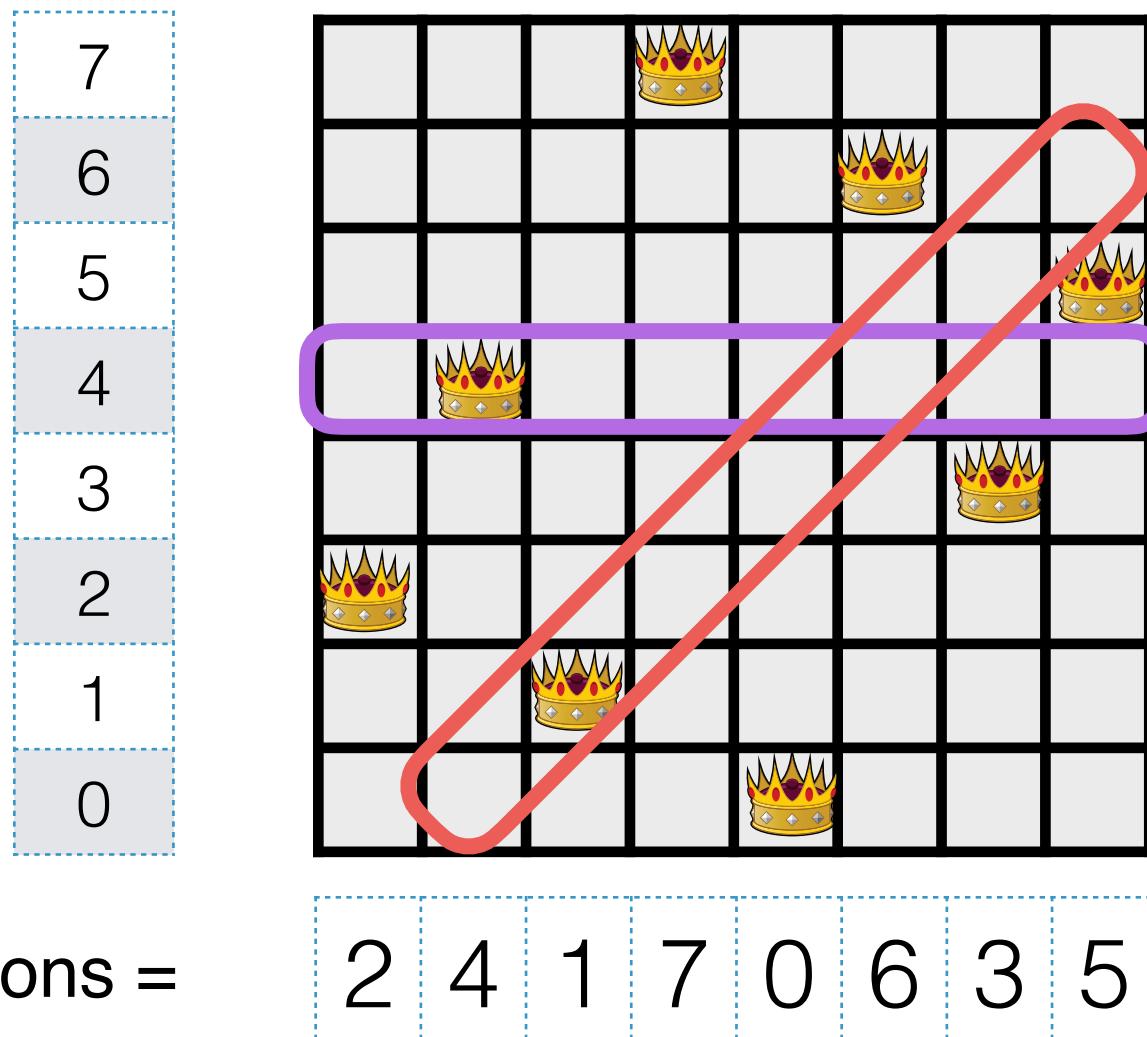
# N-Queens: modeling considerations

An integer for each column  $\{0, \dots, N-1\}$  telling in which row to place the queen



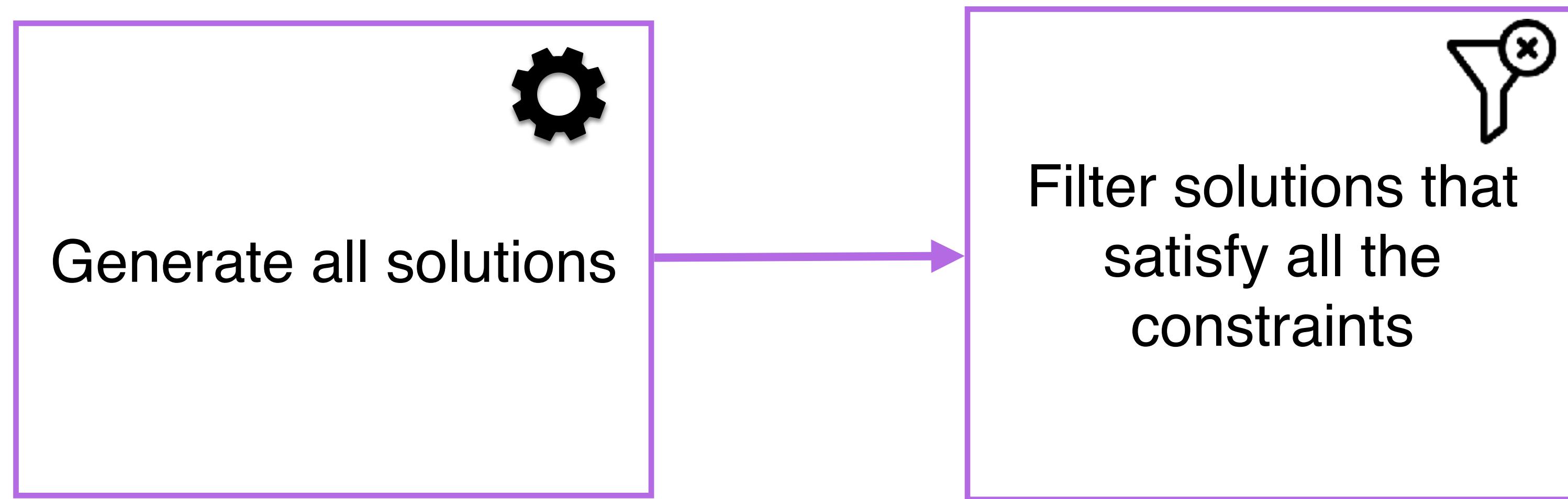
# N-Queens: modeling considerations

Advantage: only two types of constraints: no two queens share the **same row**, **column**, or **diagonal**.



# Discovering all the solutions to a CSP

- Let us make it generic



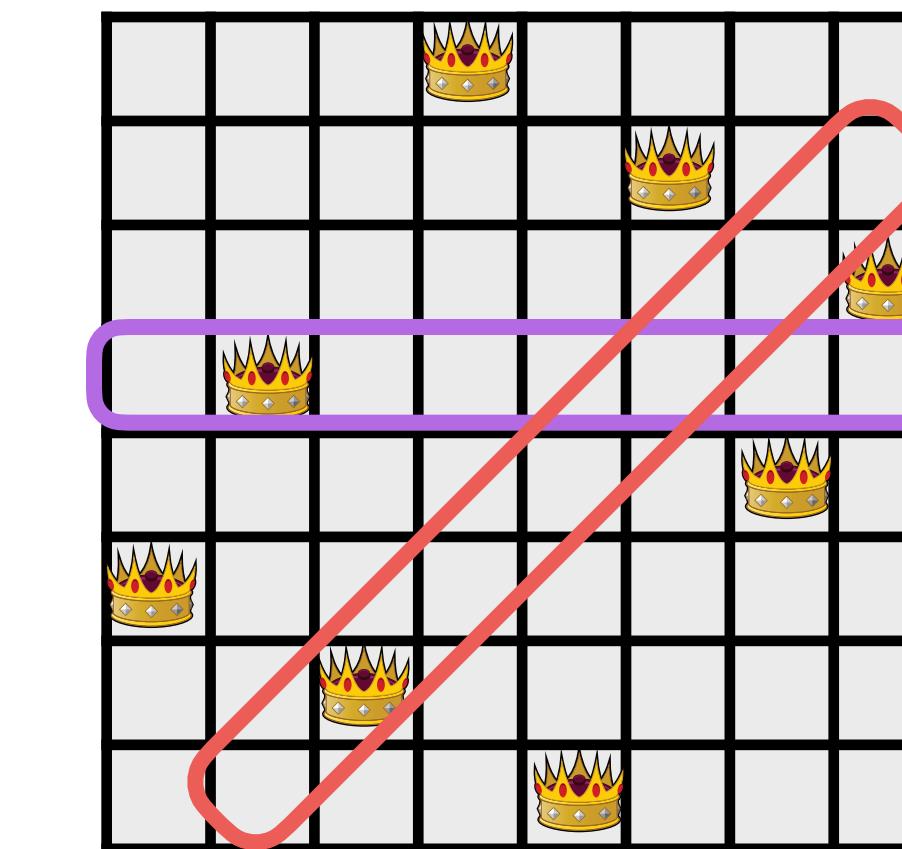
# Number of solutions in our two models

$$2^{64}$$

F	F	F	T	F	F	F	F
F	F	F	F	F	T	F	F
F	F	F	F	F	F	F	T
F	T	F	F	F	F	F	F
F	F	F	F	F	F	T	F
T	F	F	F	F	F	F	F
F	F	T	F	F	F	F	F
F	F	F	F	T	F	F	F

$$8^8 = 2^{24}$$

7
6
5
4
3
2
1
0



Decisions =

2	4	1	7	0	6	3	5
---	---	---	---	---	---	---	---

# Generate all the solutions ...

## ► Backtracking Depth First Search

```
public class NQueensChecker {  
  
    int [ ] q;  
    int n = 0;  
  
    public NQueensChecker(int n) {  
        this.n = n;  
        q = new int[n];  
    }  
  
    public void dfs() {  
        dfs(0);  
    }  
  
    private void dfs(int idx) {  
        if (idx == n) {  
            // candidate solution  
        } else {  
            for (int i = 0; i < n; i++) {  
                q[idx] = i;  
                dfs(idx+1, onSolution);  
            }  
        }  
    }  
}
```

... and filter them

## ► Backtracking Depth First Search + Filter

```
public class NQueensChecker {

    int [] q;
    int n = 0;

    public NQueensChecker(int n) {
        this.n = n;
        q = new int[n];
    }

    public void dfs() {
        dfs(0);
    }

    private void dfs(int idx) {
        if (idx == n) {
            if (constraintsSatisfied()) {
                // output solution
            }
        } else {
            for (int i = 0; i < n; i++) {
                q[idx] = i;
                dfs(idx+1);
            }
        }
    }
}
```

```
public boolean constraintsSatisfied() {
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            // no two queens on the same row
            if (q[i] == q[j]) return false;
            // no two queens on the diagonal
            if (Math.abs(q[j] - q[i]) == j-i) {
                return false;
            }
        }
    }
    return true;
}
```

Notice that this approach is quite generic.  
 You just need a method (could be made abstract) to check the constraints ✓

# "Hollywood Principle: Don't call us, we'll call you"

```

public static void main(String[ ] args) {
    NQueensChecker q = new NQueensChecker(8);
    ArrayList<int [ ]> solutions = new ArrayList<>();

    q.dfs(0, solution -> solutions.add(solution));
}

```

```

import java.util.function.Consumer;

public class NQueensChecker {

    int [ ] q;
    int n = 0;

    public NQueensChecker(int n) {
        this.n = n;
        q = new int[n];
    }

    public void dfs(Consumer<int [ ]> onSolution) {
        dfs(0, onSolution);
    }

    private void dfs(int idx, Consumer<int [ ]> onSolution) {
        if (idx == n) {
            if (constraintsSatisfied()) {
                onSolution.accept( Arrays.copyOf(q, n));
            }
        } else {
            for (int i = 0; i < n; i++) {
                q[idx] = i;
                dfs(idx+1, onSolution);
            }
        }
    }
}

```

```

@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

```

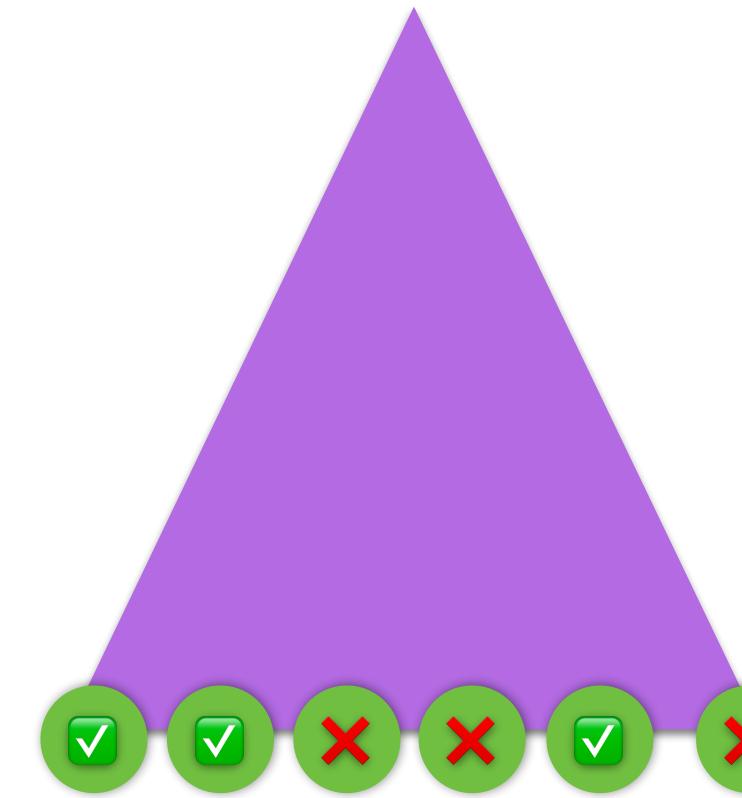
# Demo



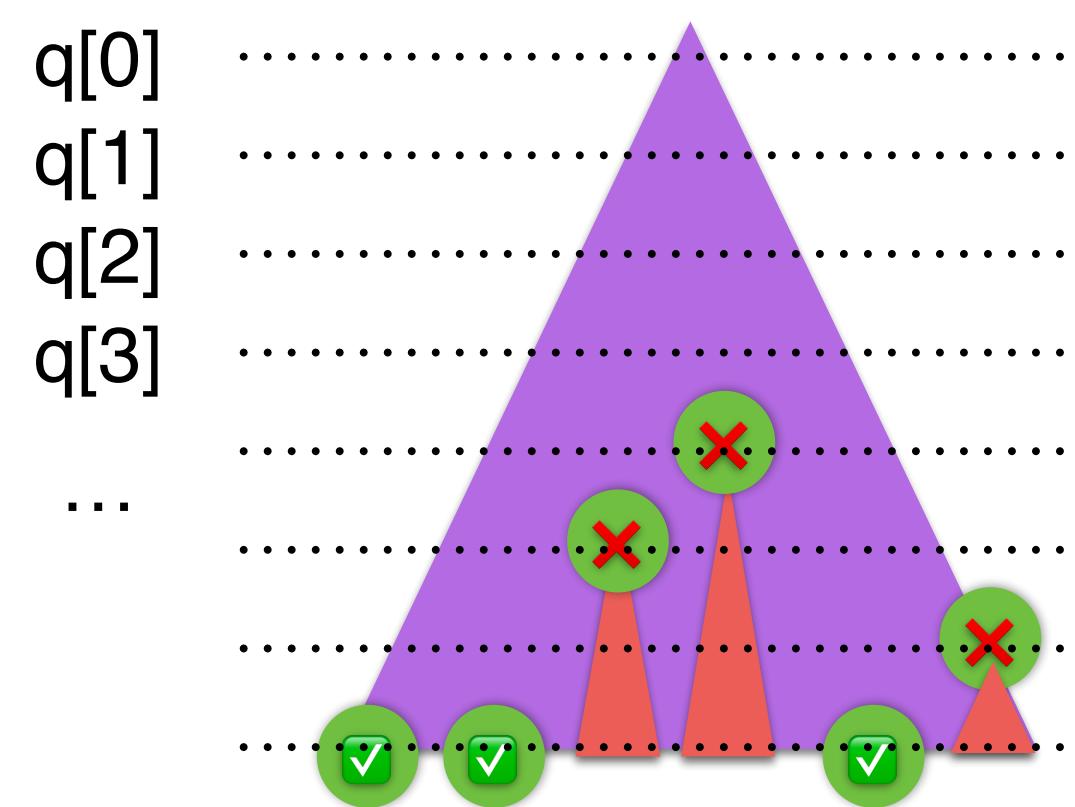
# DFS + Prune

# Principle

- DFS + filter: only verify constraints when all the decisions are finished



- DFS + Prune: verify constraints on a prefix of decisions (partial solution)



# DFS+Prune

```

public class NQueensPrune {

    int [ ] q;
    int n = 0;

    public NQueensPrune(int n) {
        this.n = n;
        q = new int[n];
    }

    public void dfs(Consumer<int [ ]> onSolution) {
        dfs(0, onSolution);
    }

    private void dfs(int idx, Consumer<int [ ]> onSolution) {
        if (idx == n) {
            onSolution.accept((Arrays.copyOf(q, n)));
        } else {
            for (int i = 0; i < n; i++) {
                q[idx] = i;
                if (constraintsSatisfied(idx))
                    dfs(idx + 1, onSolution);
            }
        }
    }
}

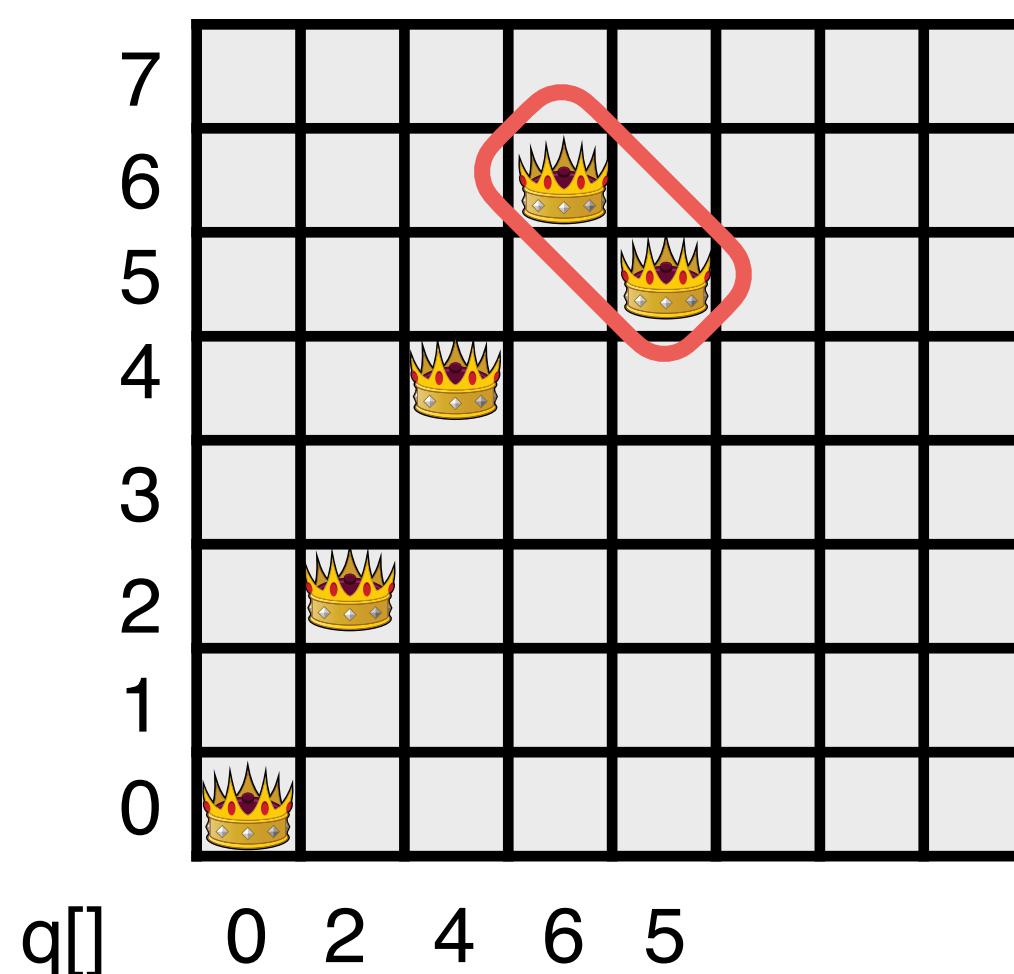
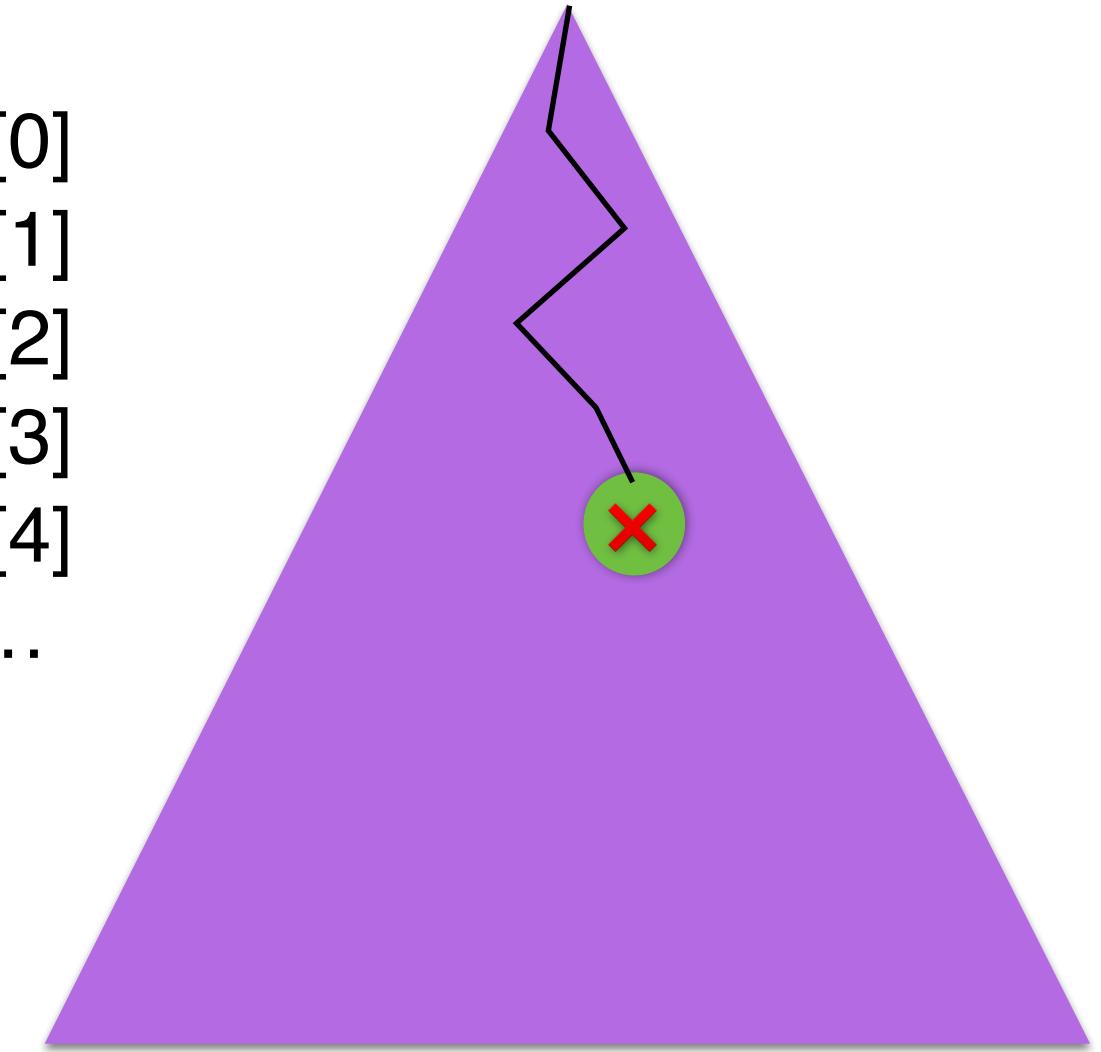
```

```

public boolean constraintsSatisfied(int j) {
    for (int i = 0; i < j; i++) {
        // no two queens on the same row
        if (q[i] == q[j]) return false;
        // no two queens on the diagonal
        if (Math.abs(q[j] - q[i]) == j - i)
            return false;
    }
    return true;
}

```

$q[0]$   
 $q[1]$   
 $q[2]$   
 $q[3]$   
 $q[4]$   
 ...



# Drawback of DFS+Prune

- ▶ Search per level
  - The backtracking works with only one index “*i*” because you overwrite previous decisions
- ▶ Only one set of decision variables
- ▶ Only one inference hardcoded and problem specific, none of the code is reusable for solving another problem, even quite similar (let’s say SUDOKU)
- ▶ Our next version will target genericity and reusability of ingredients



# Tiny-CSP Model

# N-Queens Model with Tiny-CSP

```
int n = 10;
TinyCSP csp = new TinyCSP();
Variable[] q = new Variable[n];

for (int i = 0; i < n; i++) {
    q[i] = csp.makeVariable(n);
}

for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        // queens q[i] and q[j] not on ...
        csp.notEqual(q[i],q[j],0); // ... the same line
        csp.notEqual(q[i],q[j],i-j); // ... the same left diagonal
        csp.notEqual(q[i],q[j],j-i); // ... the same right diagonal
    }
}

ArrayList<int []> solutions = new ArrayList<>();
// collect all the solutions
csp.dfs(solution -> {
    solutions.add(solution);
});
```

Variables

Constraints

Search

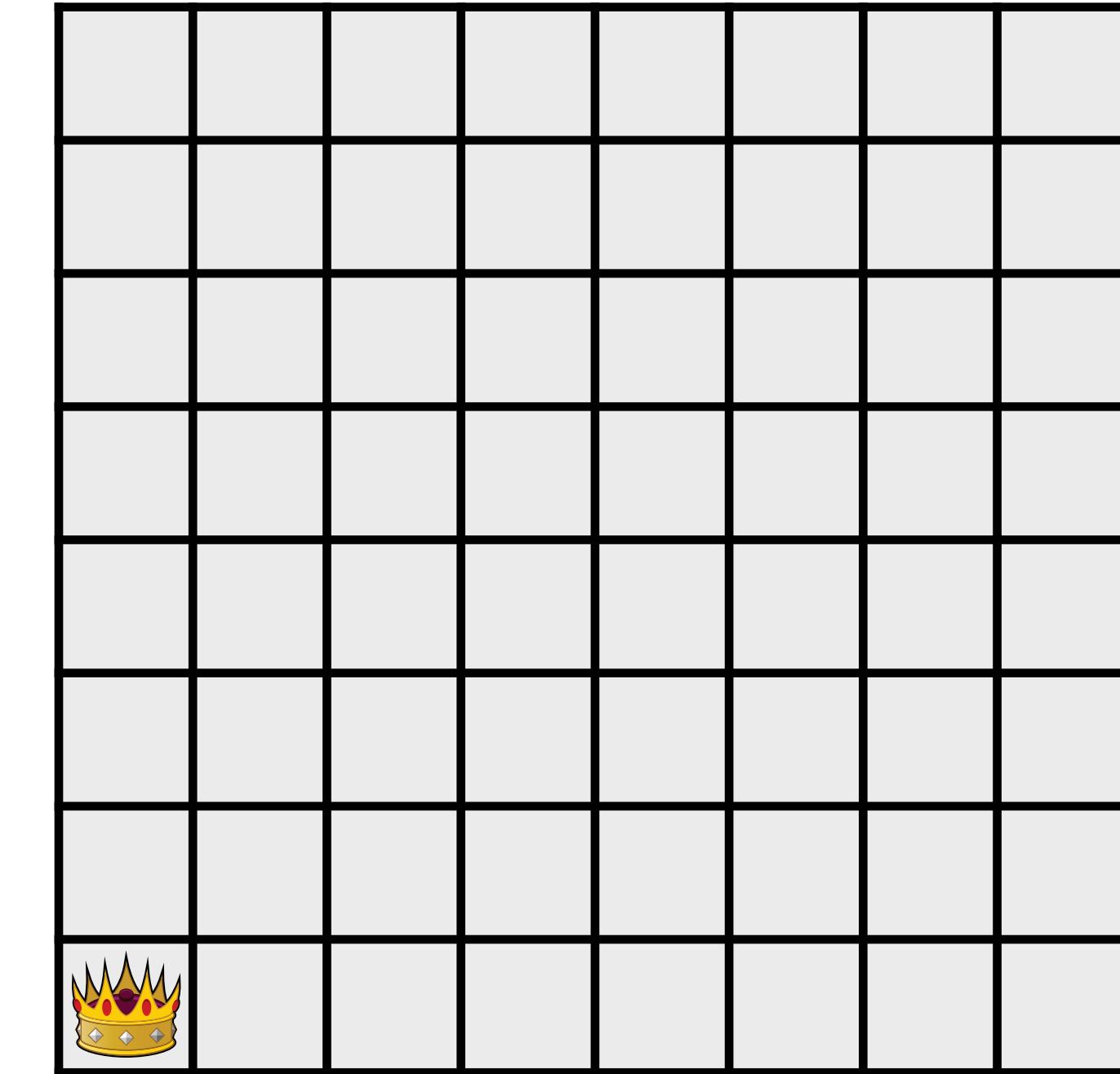
Let's make this work ...

# N-Queens: Model in MiniCP

- ▶ Representation = a model:
  - Holds an array of integer variables with one variable per column.

```
int n = 8;
TinyCSP csp = new TinyCSP();
Variable[] q = new Variable[n];

for (int i = 0; i < n; i++) {
    q[i] = csp.makeVariable(n);
}
```

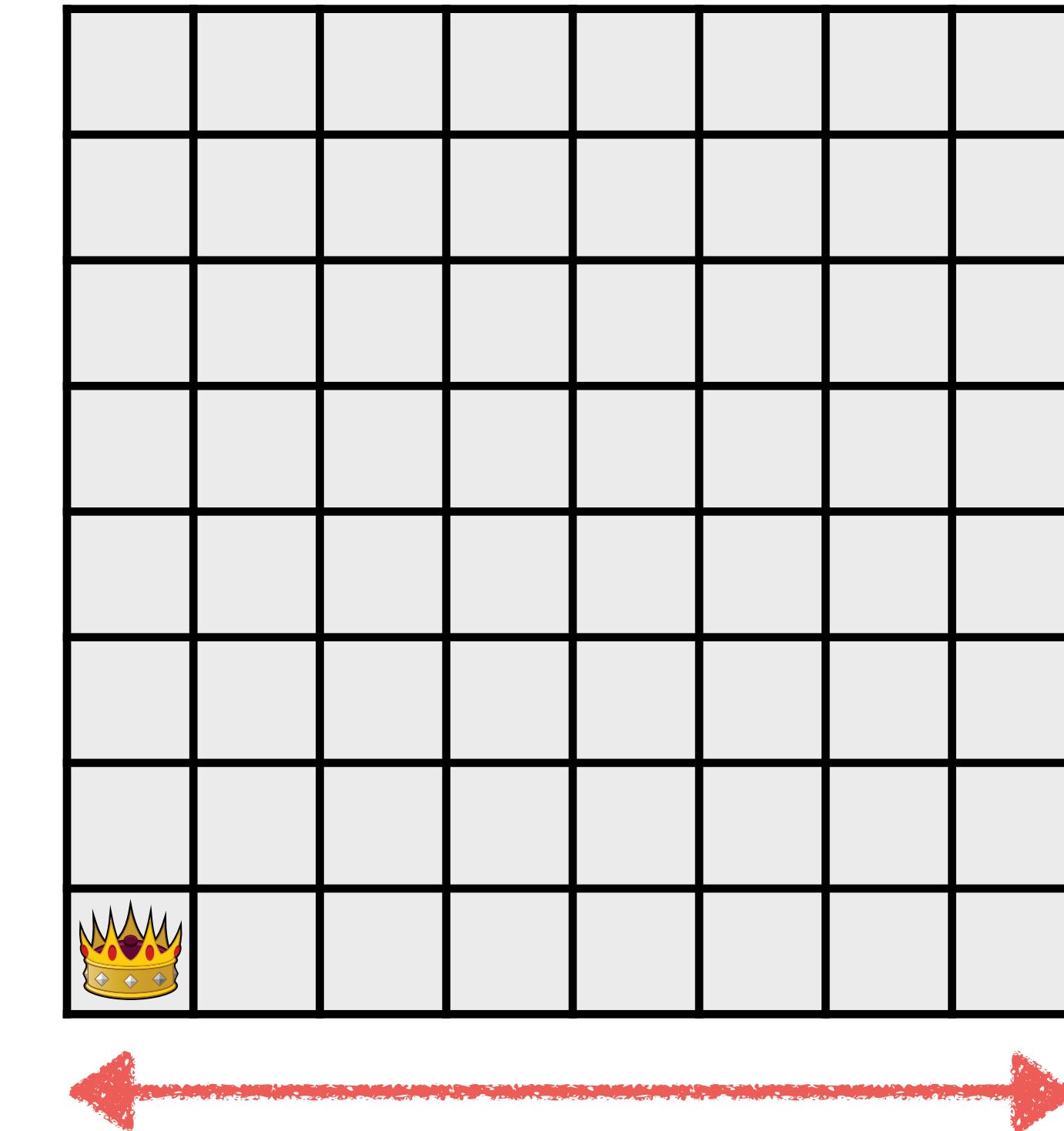


# 8-Queens: Model in MiniCP

- ▶ Representation = a model:
  - Holds an array of integer variables with one variable per column.

```
int n = 8;
TinyCSP csp = new TinyCSP();
Variable[] q = new Variable[n];

for (int i = 0; i < n; i++) {
    q[i] = csp.makeVariable(n);
}
```



# 8-Queens: Model in MiniCP

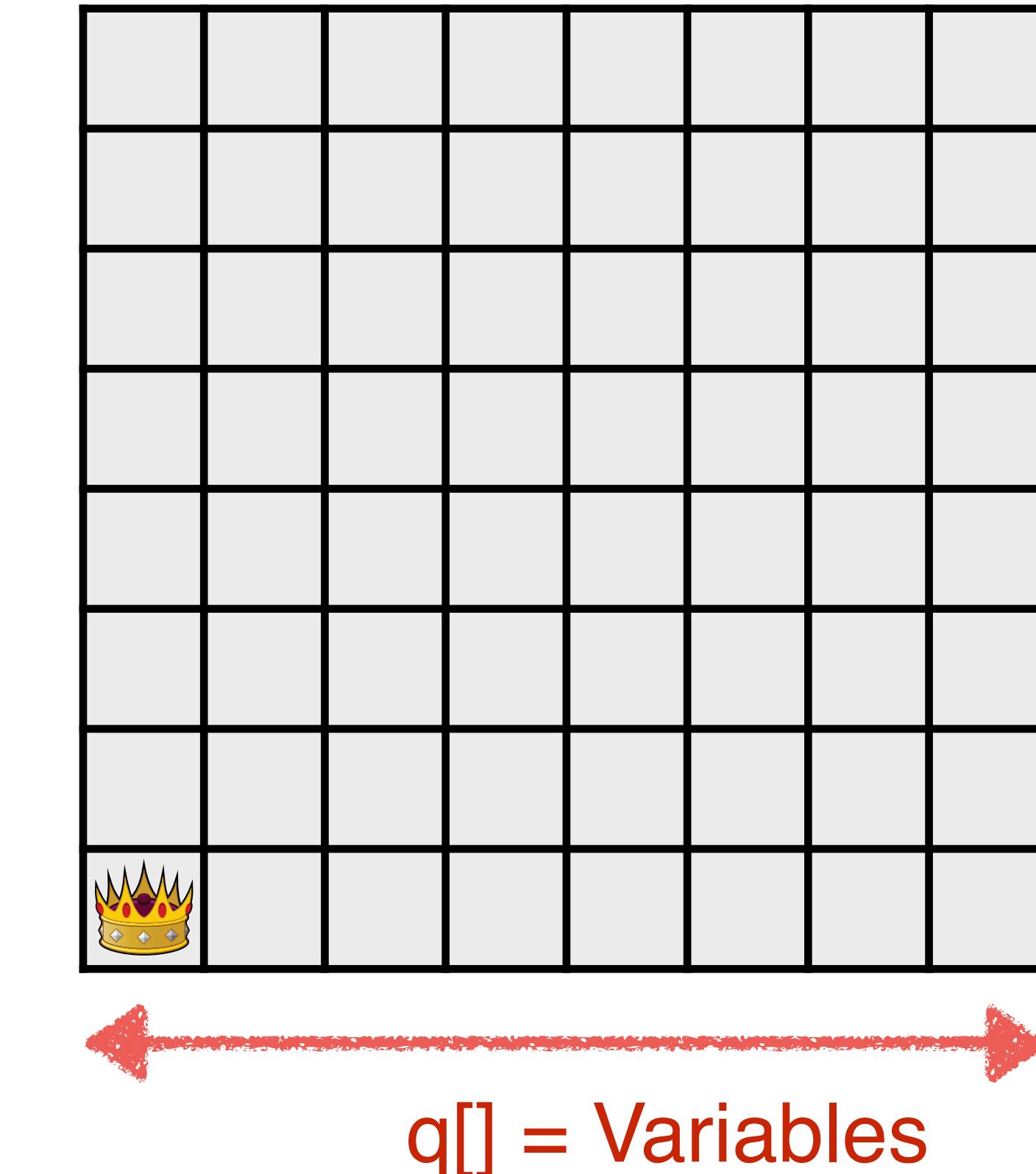
- ▶ Representation = a model:
  - Holds an array of integer variables with one variable per column.

```

int n = 8;
TinyCSP csp = new TinyCSP();
Variable[] q = new Variable[n];

for (int i = 0; i < n; i++) {
    q[i] = csp.makeVariable(n);
}
  
```

Domains  
 $D \subseteq \mathbb{Z}$



# 8-Queens: Model in MiniCP

- Representation = a model:
  - Holds an array of integer variables with one variable per column.

```

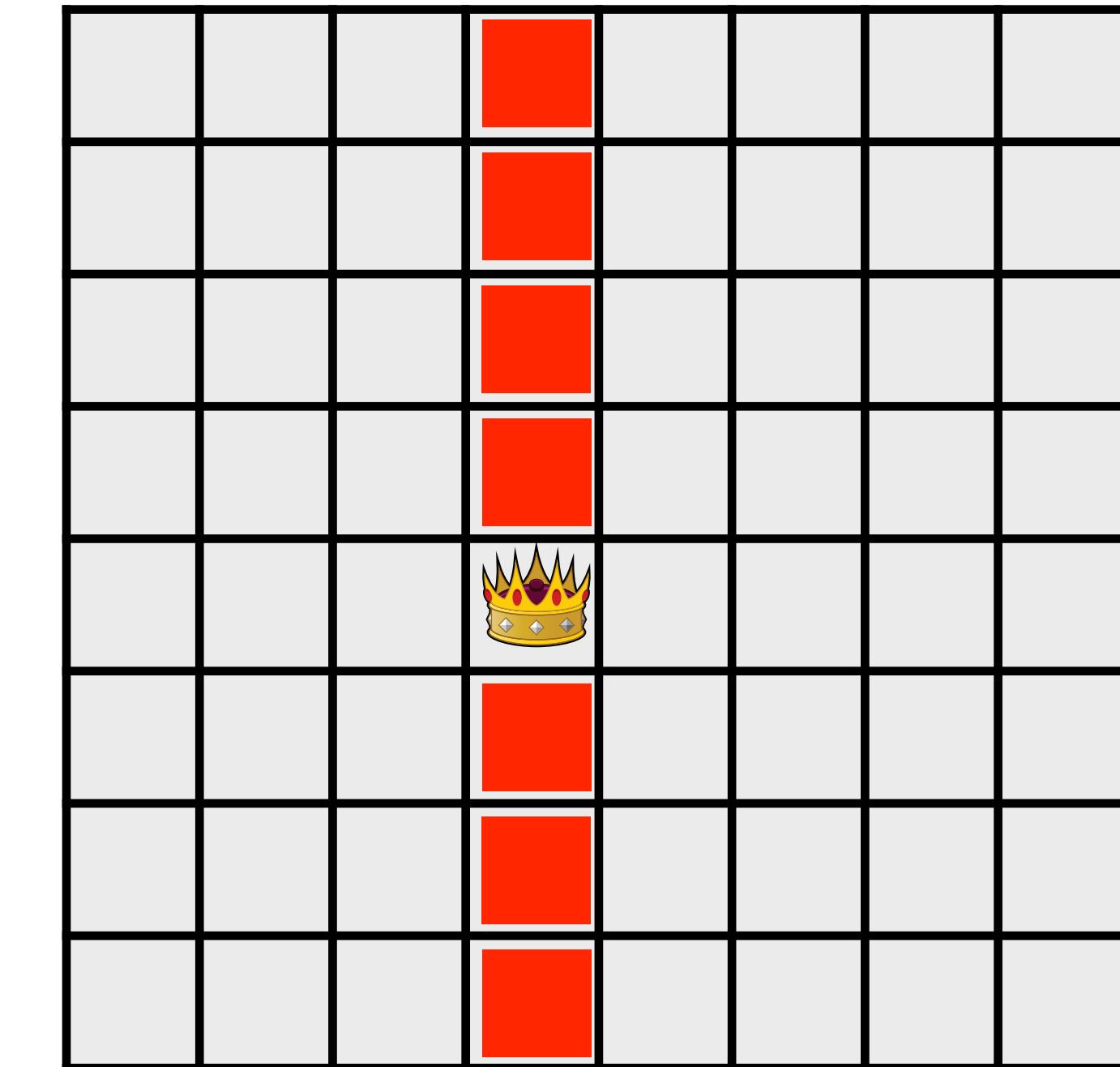
int n = 10;
TinyCSP csp = new TinyCSP();
Variable[] q = new Variable[n];

for (int i = 0; i < n; i++) {
    q[i] = csp.makeVariable(n);
}
  
```

- Cannot be in the same column...

```

for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        }
    }
  
```



# 8-Queens: Model in MiniCP

- Representation = a model:
  - Holds an array of integer variables with one variable per column

```

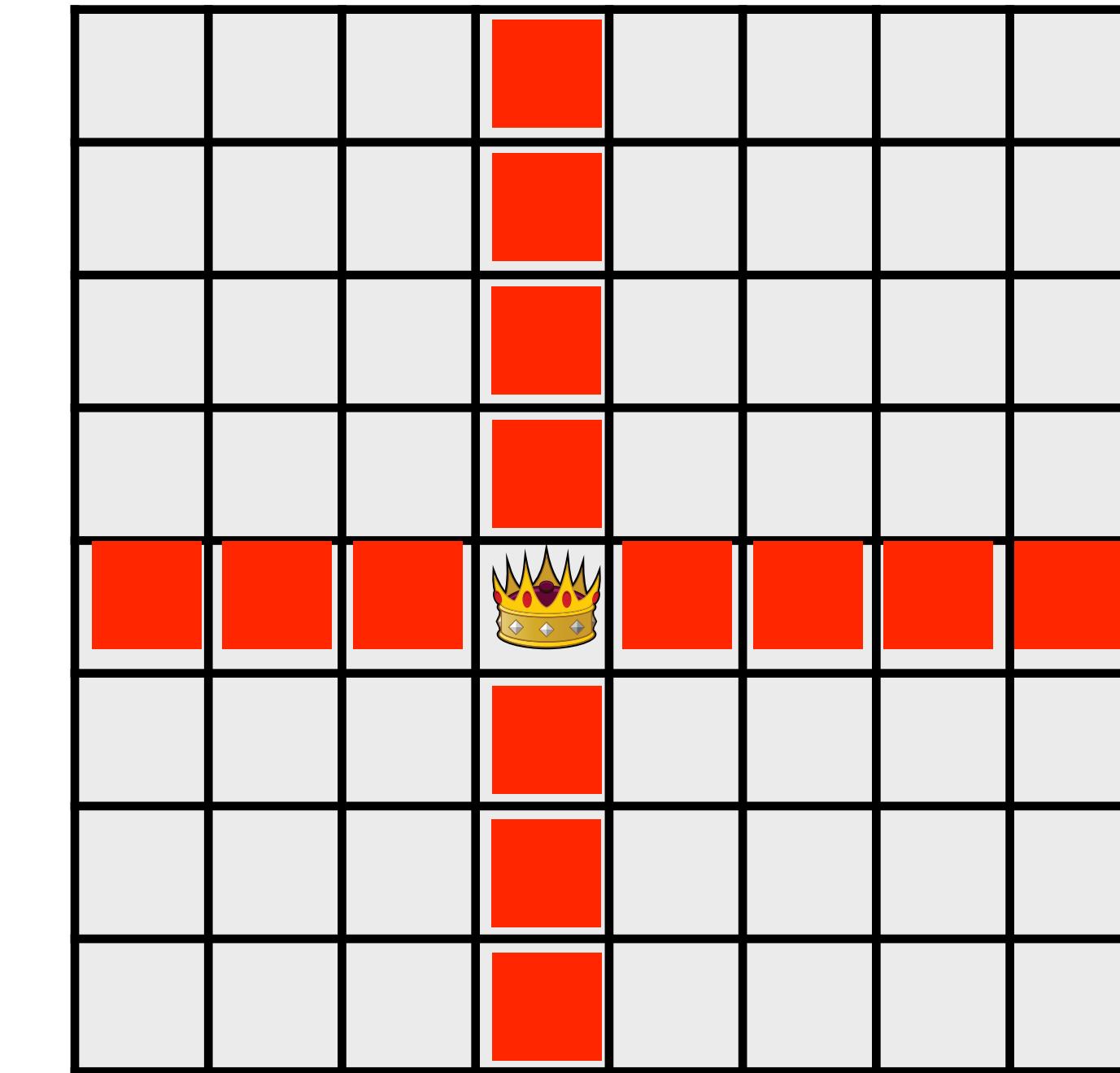
int n = 10;
TinyCSP csp = new TinyCSP();
Variable[] q = new Variable[n];

for (int i = 0; i < n; i++) {
    q[i] = csp.makeVariable(n);
}
  
```

- Cannot be on the same row...

```

for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        // queens q[i] and q[j] not on ...
        csp.notEqual(q[i],q[j],0); // line
    }
}
  
```



# 8-Queens: Model in MiniCP

- ▶ Representation = a model:
  - Holds an array of integer variables with one variable per column.

```

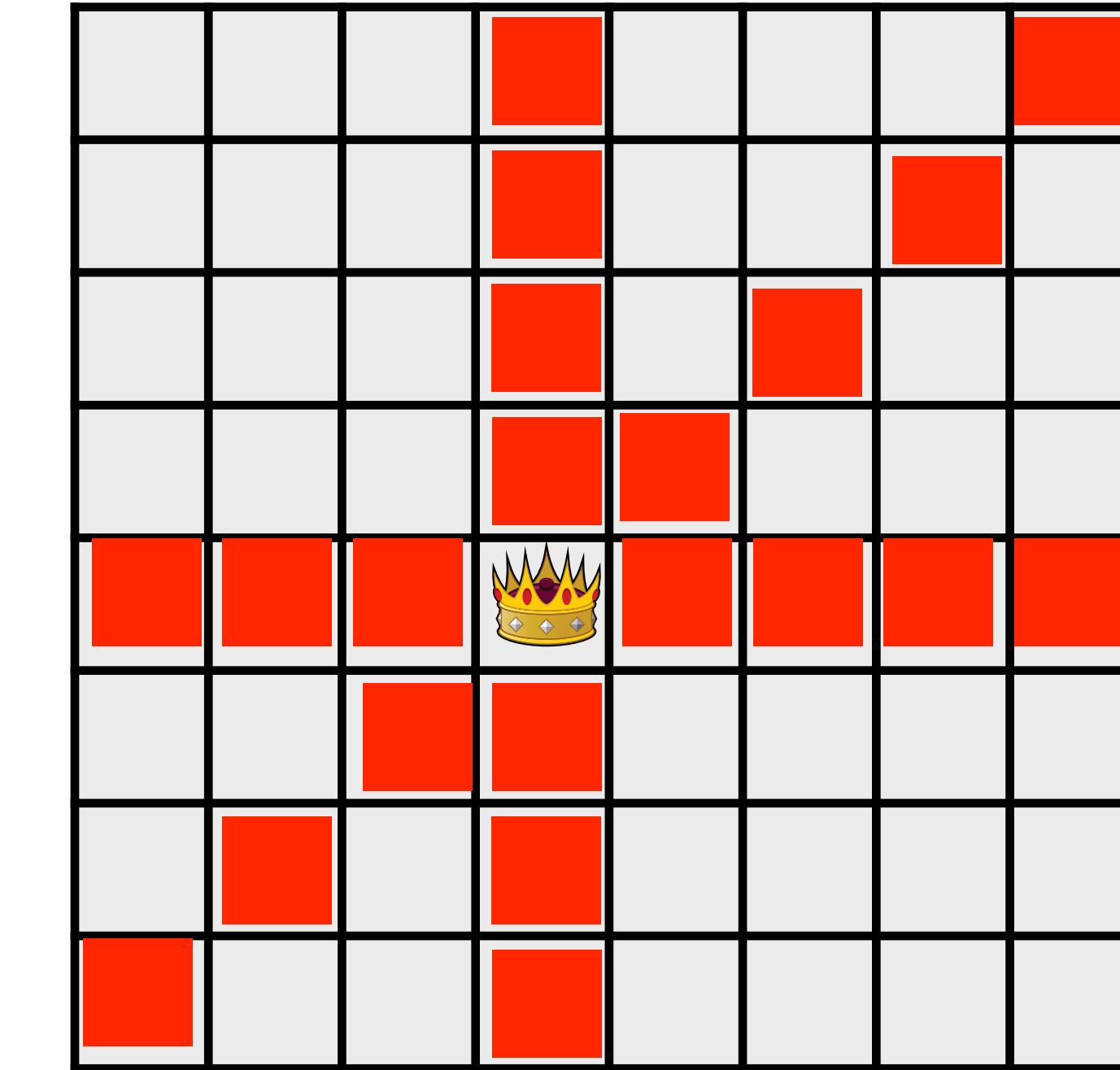
int n = 10;
TinyCSP csp = new TinyCSP();
Variable[] q = new Variable[n];

for (int i = 0; i < n; i++) {
    q[i] = csp.makeVariable(n);
}
  
```

- Cannot be on the same diagonal...

```

for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        // queens q[i] and q[j] not on ...
        csp.notEqual(q[i],q[j],0); // line
        csp.notEqual(q[i],q[j],i-j); // left diagonal
    }
}
  
```



# 8-Queens: Model in MiniCP

- Representation = a model:
  - Holds an array of integer variables with one variable per column.

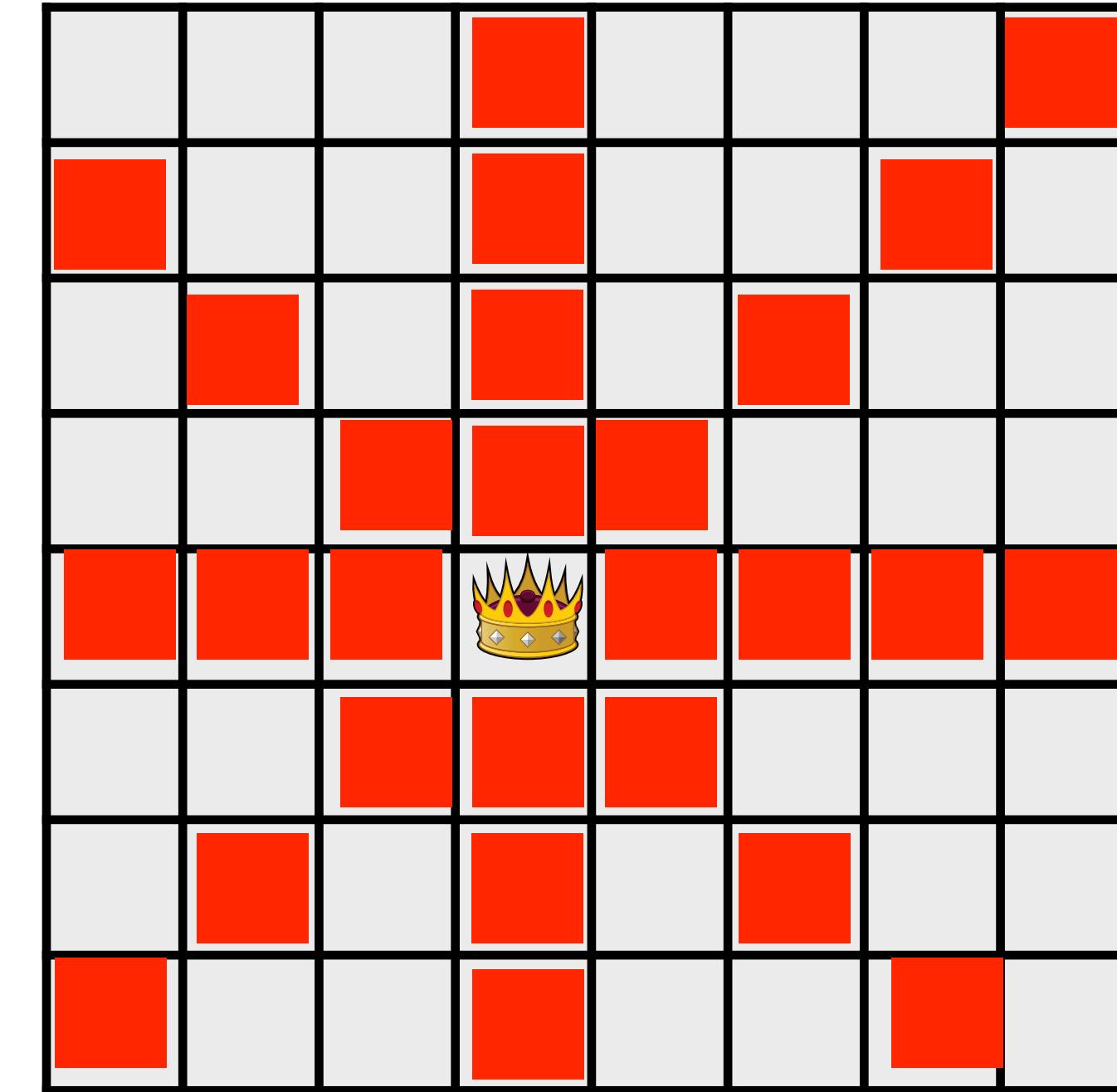
```

int n = 10;
TinyCSP csp = new TinyCSP();
Variable[] q = new Variable[n];

for (int i = 0; i < n; i++) {
    q[i] = csp.makeVariable(n);
}

for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        // queens q[i] and q[j] not on ...
        csp.notEqual(q[i],q[j],0); // line
        csp.notEqual(q[i],q[j],i-j); // left diagonal
        csp.notEqual(q[i],q[j],j-i); // right diagonal
    }
}

```



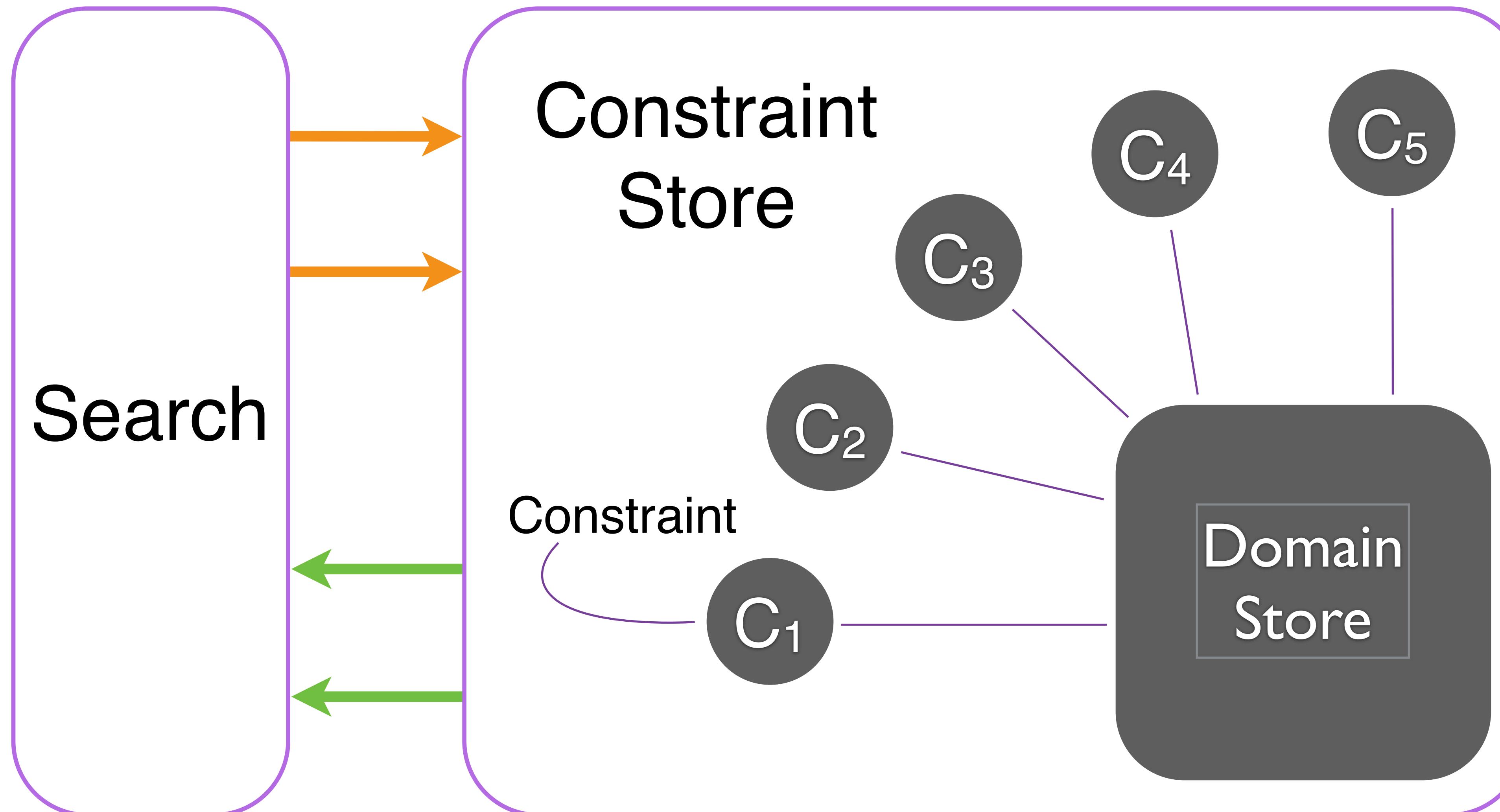
- Cannot be on the same diagonals...



# Tiny-CSP

# Computation

# Computational Paradigm



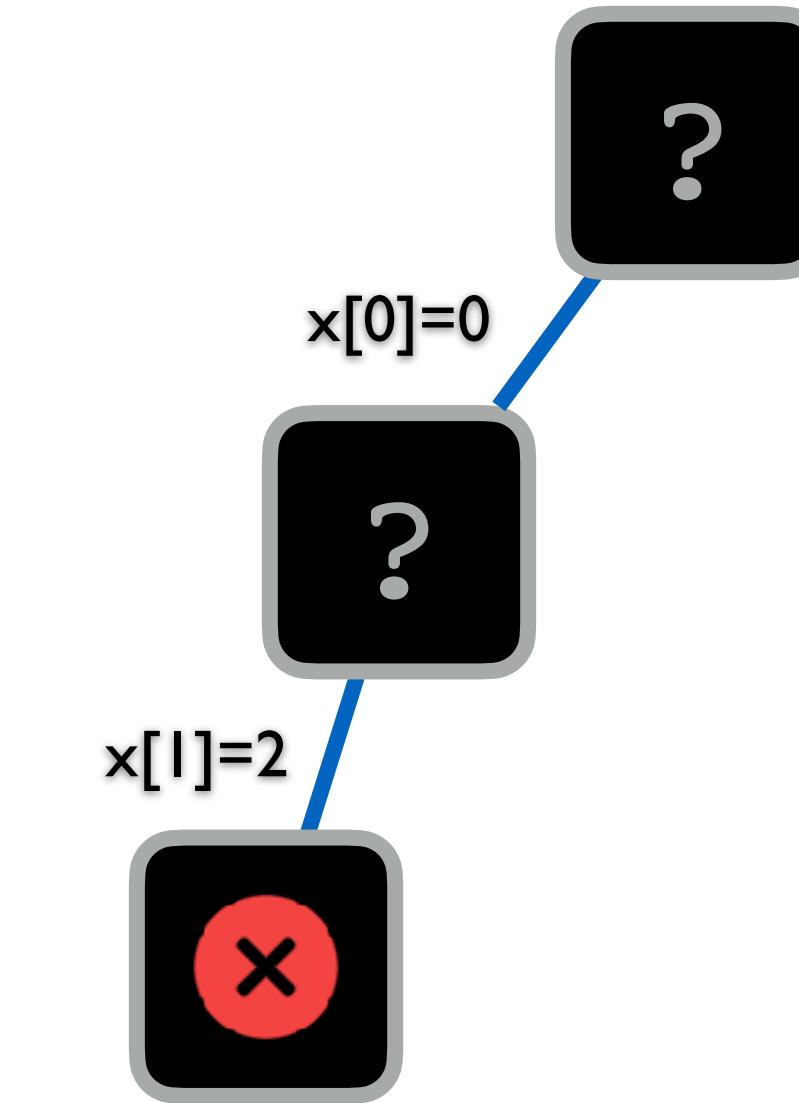
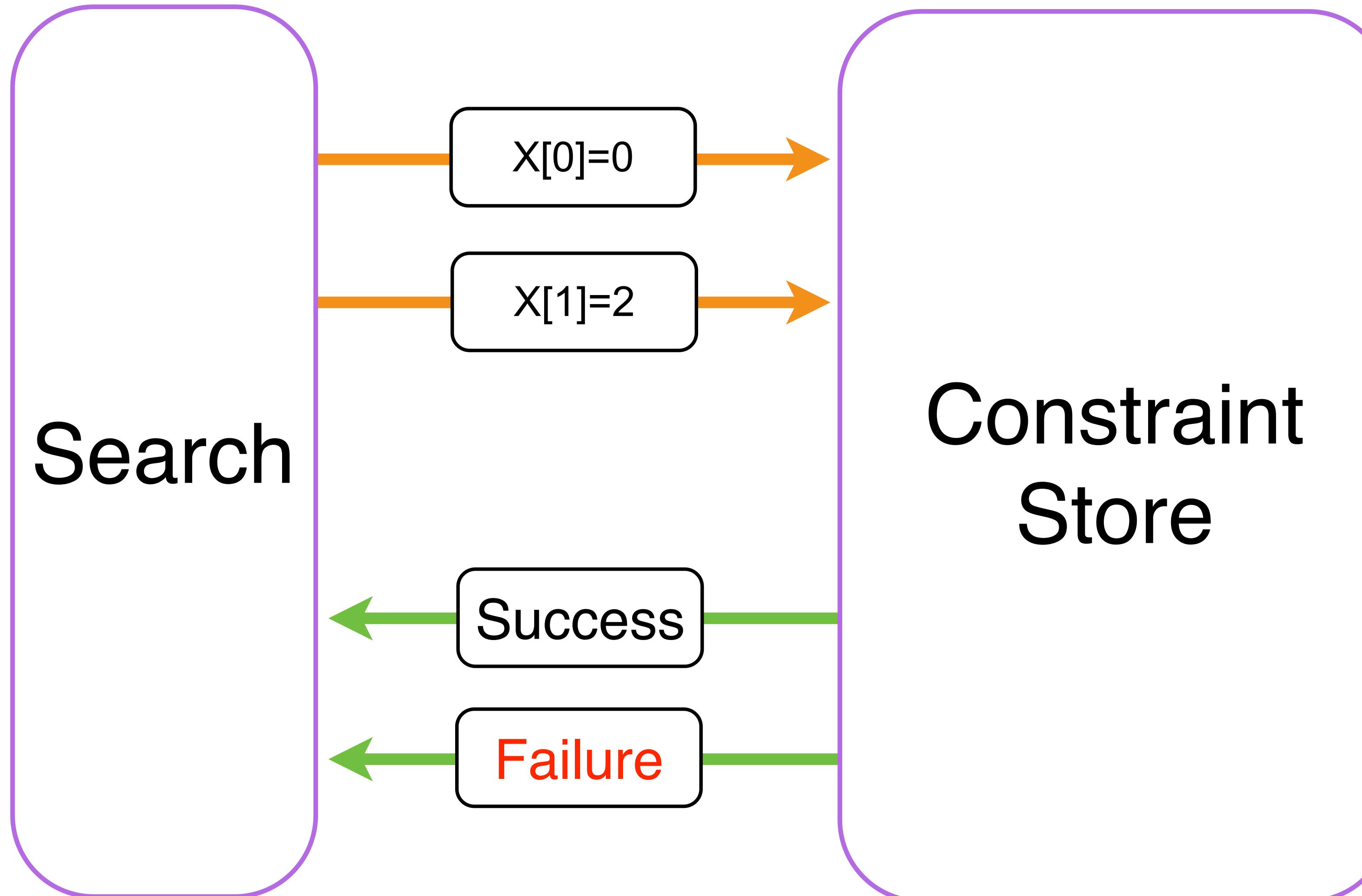
# Computational Paradigm

The propagation engine:

- This is the core of any constraint-programming solver.
- It is a simple fixpoint algorithm:

```
fixPoint()
{
    repeat
        select a constraint c;
        if c is infeasible given the domain store then
            return failure;
        else
            apply the pruning algorithm associated with c;
    until (no constraint can remove any value from the domain of its variables);
    return success;
}
```

# Computational Paradigm



# TinyCSP class

## Constraint Store

```

public class TinyCSP {

    List<Constraint> constraints = new LinkedList<>();
    List<Variable> variables = new LinkedList<>();

    public Variable makeVariable(int domSize) {
        Variable x = new Variable(domSize);
        variables.add(x);
        return x;
    }

    public void notEqual(Variable x, Variable y, int offset) {
        constraints.add(new NotEqual(x, y, offset));
        fixPoint();
    }

    public void fixPoint() {
        boolean fix = false;
        while (!fix) {
            fix = true;
            for (Constraint c : constraints) {
                fix &= !c.propagate();
            }
        }
    }
}

```

```

abstract class Constraint {
    /**
     * Propagate the constraint and return
     * true if any value could be removed
     * @return true if at least one value of one
     *         variable could be removed
     */
    abstract boolean propagate();
}

public class Variable {
    Domain dom;

    /**
     * Creates a variable with domain {0..n-1}
     */
    public Variable(int n) {
        dom = new Domain(n);
    }
}

```

# What does a constraint do ?

- ▶ Feasibility checking:
  - Can the constraint be satisfied given the values in the domains of its variables?
- ▶ Pruning:
  - If satisfiable = feasible, then a constraint removes values in the domains that cannot be part of any solution.

# The Not Equal Constraint $x \neq y + \text{offset}$

```
class NotEqual extends Constraint {

    Variable x, y;
    int offset;

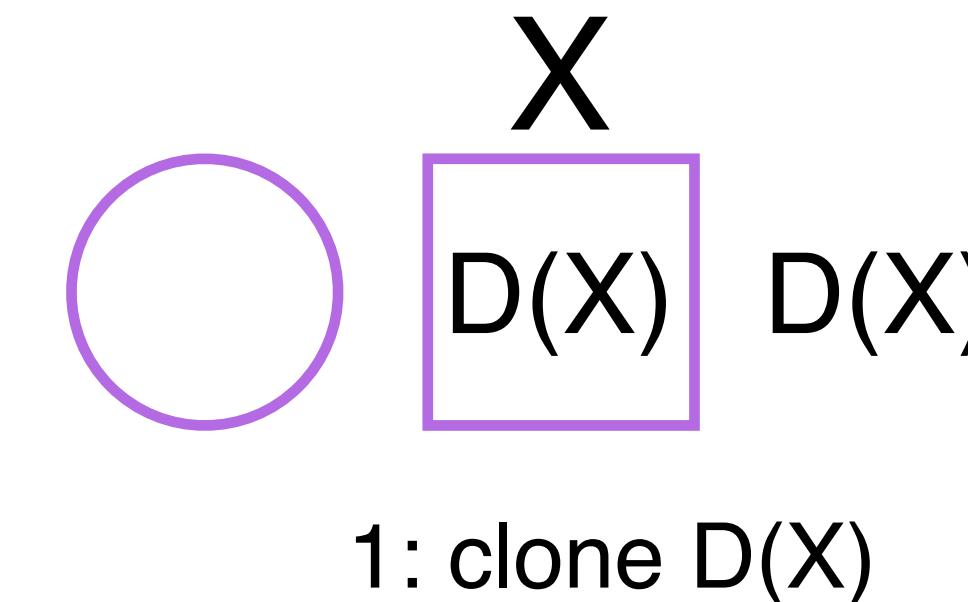
    public NotEqual(Variable x, Variable y, int offset) {
        this.x = x;
        this.y = y;
        this.offset = offset;
    }

    public NotEqual(Variable x, Variable y) {
        this(x, y, 0);
    }

    @Override
    boolean propagate() {
        if (x.dom.isFixed()) {
            return y.dom.remove(x.dom.min() - offset);
        }
        if (y.dom.isFixed()) {
            return x.dom.remove(y.dom.min() + offset);
        }
        return false;
    }
}
```

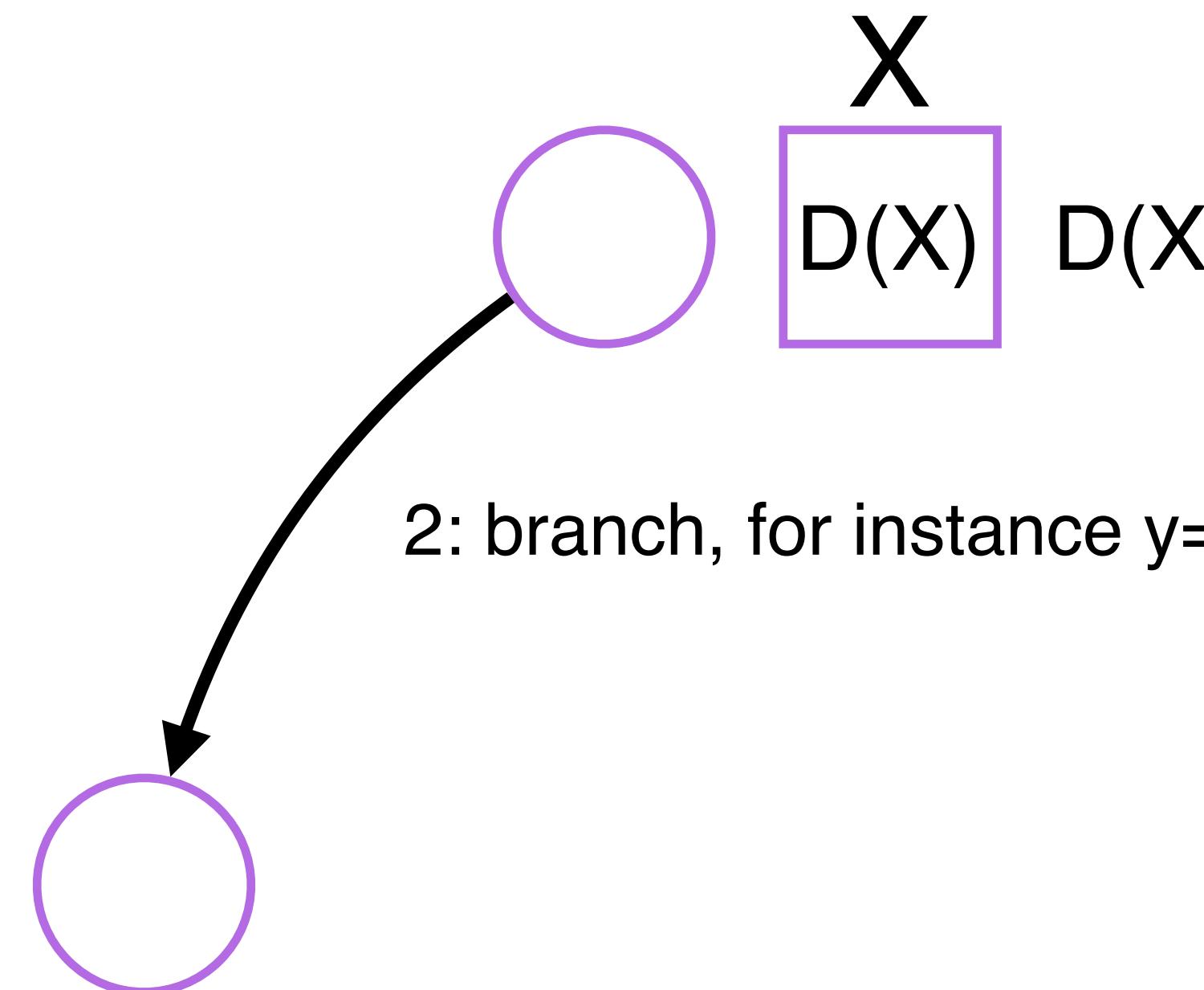
# State Management

- When a value is removed it needs to be restored on backtrack
- TinyCSP will use a “backup” mechanism of the domains



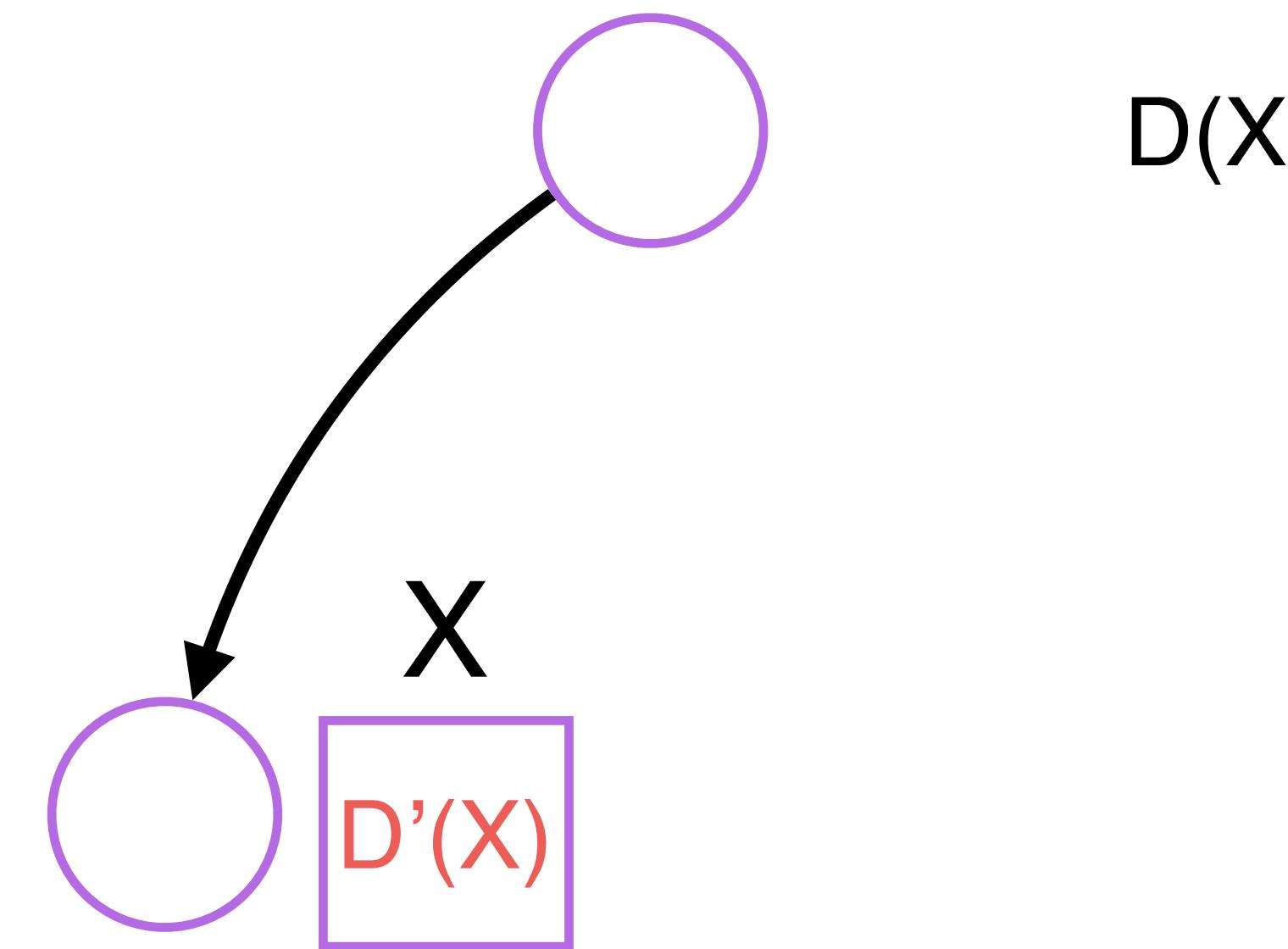
# State Management

- When a value is removed it needs to be restored on backtrack
- TinyCSP will use a “backup” mechanism of the domains



# State Management

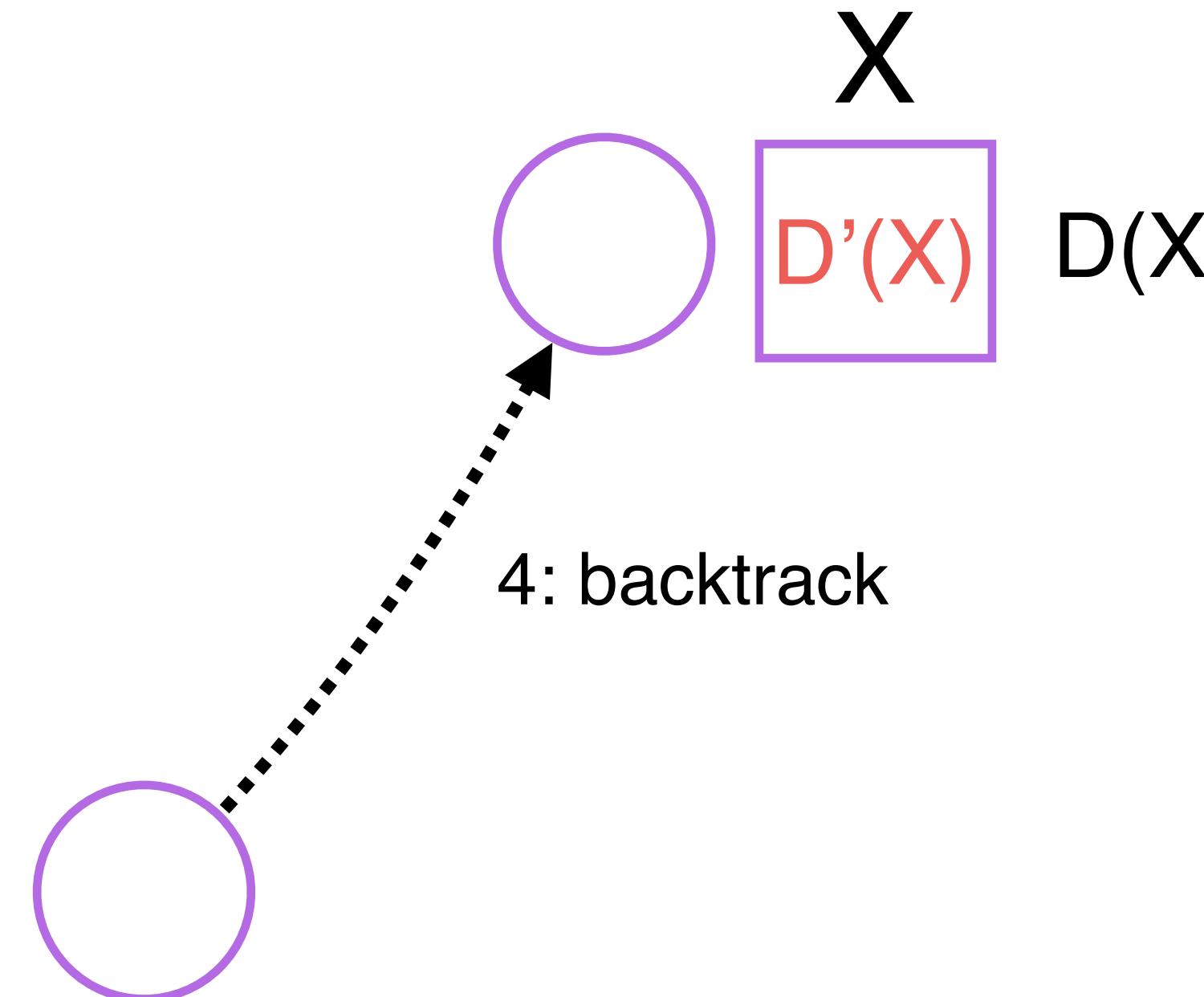
- When a value is removed it needs to be restored on backtrack
- TinyCSP will use a “backup” mechanism of the domains



3: fix-point,  $D(X)$  may be modified

# State Management

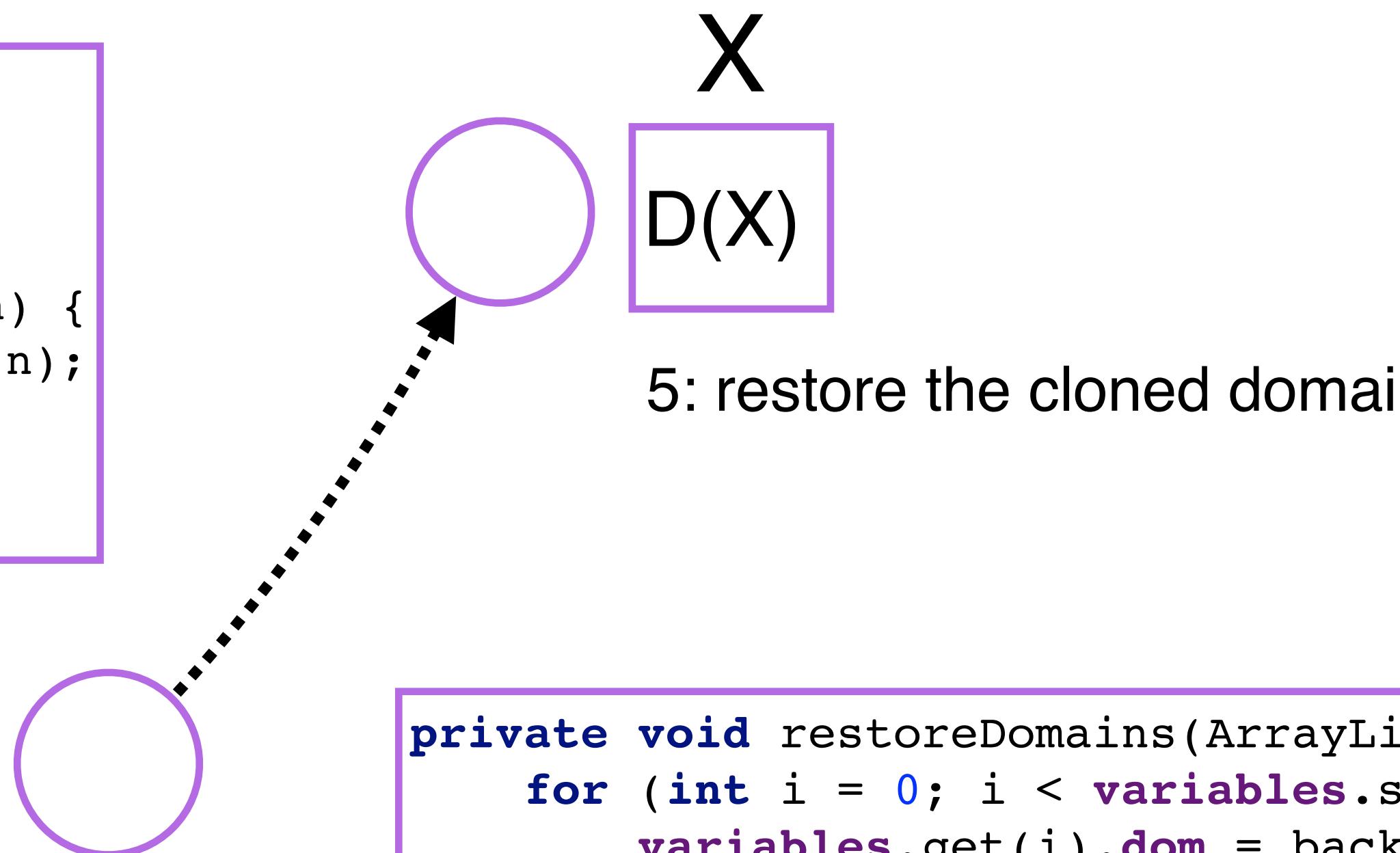
- When a value is removed it needs to be restored on backtrack
- TinyCSP will use a “backup” mechanism of the domains



# State Management

- When a value is removed it needs to be restored on backtrack
- TinyCSP will use a “backup” mechanism of the domains

```
public class Variable {  
  
    Domain dom;  
  
    public Variable(int n) {  
        dom = new Domain(n);  
    }  
}
```



```
private void restoreDomains(ArrayList<Domain> backup) {  
    for (int i = 0; i < variables.size(); i++) {  
        variables.get(i).dom = backup.get(i);  
    }  
}
```

```
public void dfs(Consumer<int[]> onSolution) {
    // pickup a variable that is not yet fixed if any
    Optional<Variable> notFixed = firstNotFixed();
    if (!notFixed.isPresent()) { // all variables fixed, a solution is found
        int[] solution = variables.stream().mapToInt(x -> x.dom.min()).toArray();
        onSolution.accept(solution);
    } else {
        Variable y = notFixed.get(); // take the unfixed variable
        int v = y.dom.min();
        ArrayList<Domain> backup = backupDomains();
        // left branch x = v
        try {
            y.dom.fix(v);
            fixPoint();
            dfs(onSolution);
        } catch (Inconsistency i) {
        }
        restoreDomains(backup);
        // right branch x != v
        try {
            y.dom.remove(v);
            fixPoint();
            dfs(onSolution);
        } catch (Inconsistency i) {
        }
    }
}
```

Clone domains

Branch (left) and Fix-Point

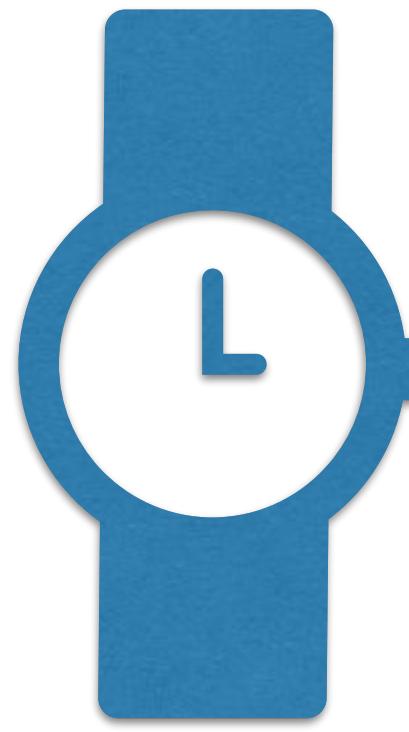
Restore domains

Branch (right) and Fix-Point

# Domain implementation: java.util.BitSet

```
public class Domain {  
  
    private BitSet values;  
  
    public Domain(int n) {  
        values = new BitSet(n);  
        values.set(0, n);  
    }  
  
    public boolean isFixed() { size() == 1; }  
    public int size() { return values.cardinality(); }  
    public int min() { return values.nextSetBit(0); }  
  
    public boolean remove(int v) {  
        if (0 <= v && v < values.length()) {  
            if (values.get(v)) {  
                values.clear(v);  
                if (size() == 0) throw new TinyCSP.Inconsistency();  
                return true;  
            }  
        }  
        return false;  
    }  
  
    public void fix(int v) {  
        if (!values.get(v)) throw new TinyCSP.Inconsistency();  
        values.clear();  
        values.set(v);  
    }  
  
    public Domain clone() {  
        return new Domain((BitSet) values.clone());  
    }  
}
```

# Performances



# What to measure ?

- ▶ The number of nodes (recursive calls)
- ▶ The time
- ▶ Let's compare the three approaches
  - NQueensChecker (generate and filter)
  - NQueensPrune (prune the search when violation detected on prefix of decisions)
  - NQueensTinyTSP (using the tiny CSP solver)

# NQueensChecker

<b>N</b>	<b>Nodes</b>	<b>Time (ms)</b>	<b>#solutions</b>
8	19173961	167	92
9	435.848.050	4.526	352
10	11.111.111.111	101.497	724

# NQueensPrune

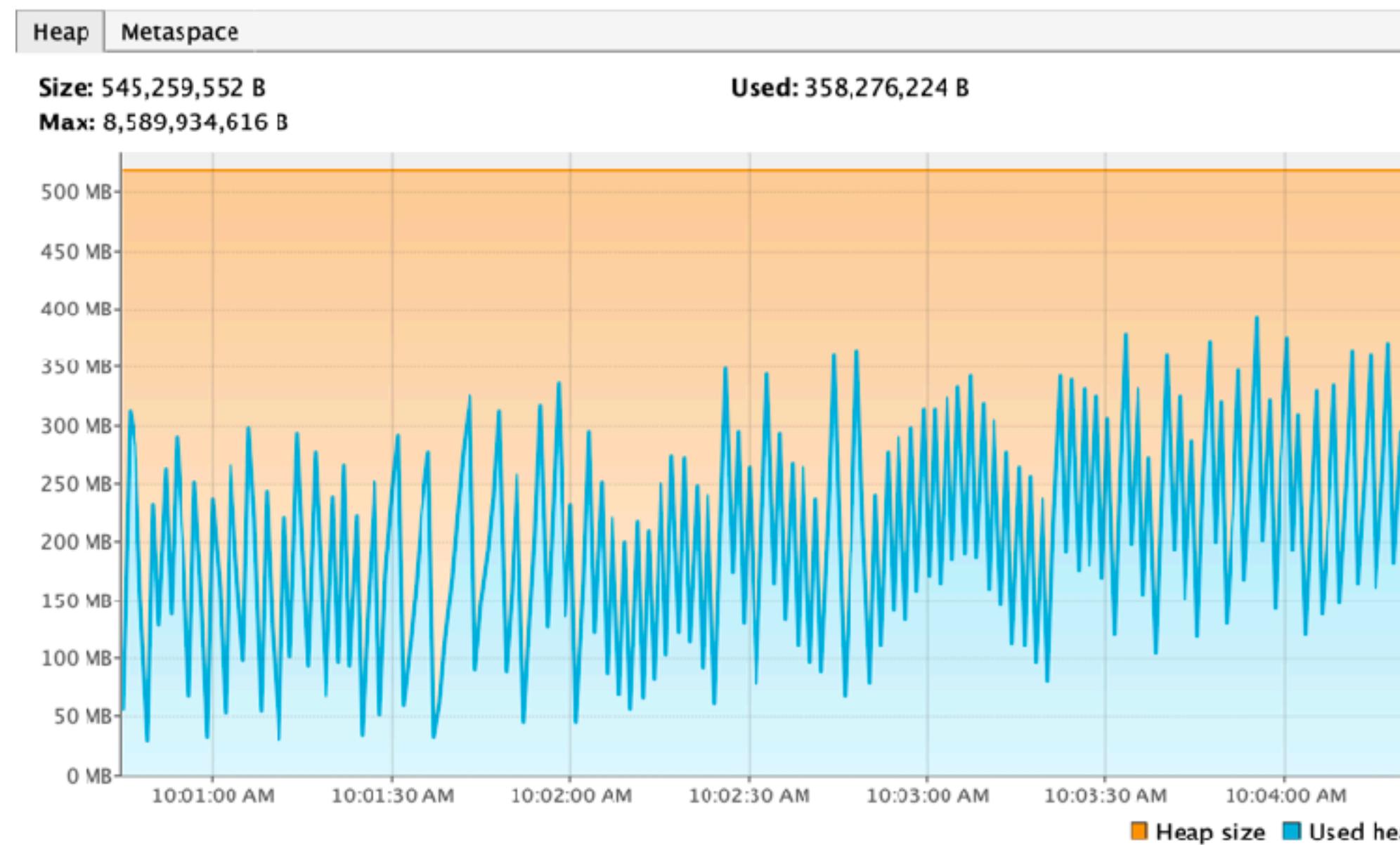
<b>N</b>	<b>Nodes</b>	<b>Time (ms)</b>	<b>#solutions</b>
12	856189	130	14200
13	4674890	690	73712
14	27358553	4550	365596
15	171129072	30138	2279184

# NQueensTinyCSP

<b>N</b>	<b>Nodes</b>	<b>Time (ms)</b>	<b>#solutions</b>
12	102531	2439	14200
13	73712	11999	73712
14	2934559	72753	365596
15	17543706	477324	2279184

# Where do we loose time in NQueensTinyCSP ?

- Profiler (Visual VM <https://visualvm.github.io>)



Name	Total Time (CPU)	Total Time
tinycsp.examples.NQueensTinyCSP.main ()	152,296 ms (23.3%)	152,296 ms (23.3%)
tinycsp.TinyCSP.dfs ()	152,296 ms (23.3%)	152,296 ms (23.3%)
tinycsp.TinyCSP.fixPoint ()	67,361 ms (10.3%)	67,361 ms (10.3%)
tinycsp.TinyCSP\$Inconsistency.<init> ()	59,579 ms (9.1%)	59,579 ms (9.1%)
tinycsp.Domain.fix ()	53,956 ms (8.2%)	53,956 ms (8.2%)
tinycsp.NotEqual.propagate ()	48,920 ms (7.5%)	48,920 ms (7.5%)
tinycsp.Domain.size ()	41,484 ms (6.3%)	41,484 ms (6.3%)
tinycsp.Domain.isFixed ()	41,381 ms (6.3%)	41,381 ms (6.3%)
tinycsp.TinyCSP.backupDomains ()	16,493 ms (2.5%)	16,493 ms (2.5%)
tinycsp.TinyCSP.restoreDomains ()	9,710 ms (1.5%)	9,710 ms (1.5%)
tinycsp.Domain.remove ()	8,032 ms (1.2%)	8,032 ms (1.2%)
tinycsp.TinyCSP.firstNotFixed ()	1,591 ms (0.2%)	1,591 ms (0.2%)
tinycsp.TinyCSP.lambda\$firstNotFixed\$0 ()	493 ms (0.1%)	493 ms (0.1%)
tinycsp.TinyCSP\$\$Lambda\$17.0x0000000800c03458.test ()	493 ms (0.1%)	493 ms (0.1%)

# One source of inefficiency: The Fixpoint Algorithm

```

fixPoint()
{
    repeat
        select a constraint  $c$ ;
        if  $c$  is infeasible given the domain store then
            return failure;
        else
            apply the pruning algorithm associated with  $c$ ;
    until (no constraint can remove any value);
    return success;
}

```

**Data:** The CSP  $\langle X, \mathcal{D}^0, C \rangle$   
**Result:** The greatest fixpoint domain  
 $pruningNeeded \leftarrow true$   
 $\mathcal{D} \leftarrow \mathcal{D}^0$   
**while**  $pruningNeeded$  **do**  
     $\mathcal{D}^p \leftarrow \mathcal{F}_C(\mathcal{D})$   
     $pruningNeeded \leftarrow \mathcal{D}^p \neq \mathcal{D}$   
     $\mathcal{D} \leftarrow \mathcal{D}^p$   
**end**



If no domain of a variable of the constraint  $c$  was changed since last time it was executed, is it worth executing it again?

# Improved Fixpoint Algorithm: Data-Driven

- The first algorithm is “naïve”:
  - It invokes  $\mathcal{F}_c$  on every constraint  $c$  all the time.
- We can make this far better!



**Data:** a CSP  $\langle X, D^0, C \rangle$

**Result:** the greatest fixpoint of the filtering algorithms for the constraints in  $C$ , starting from the domains  $D^0$  of the variables of  $X$

```

 $Q \leftarrow C$ 
 $D \leftarrow D^0$ 
while  $|Q| > 0$  do
   $c \leftarrow \text{dequeue}(Q)$ 
   $D' \leftarrow \mathcal{F}_c(D)$ 
   $V \leftarrow \{x \in \text{Vars}(c) : D'(x) \neq D(x)\}$ 
  if  $|V| > 0$  then
     $\quad \text{enqueue}(Q, \{c' \in C : |\text{Vars}(c') \cap V| > 0\})$ 
   $D \leftarrow D'$ 

```

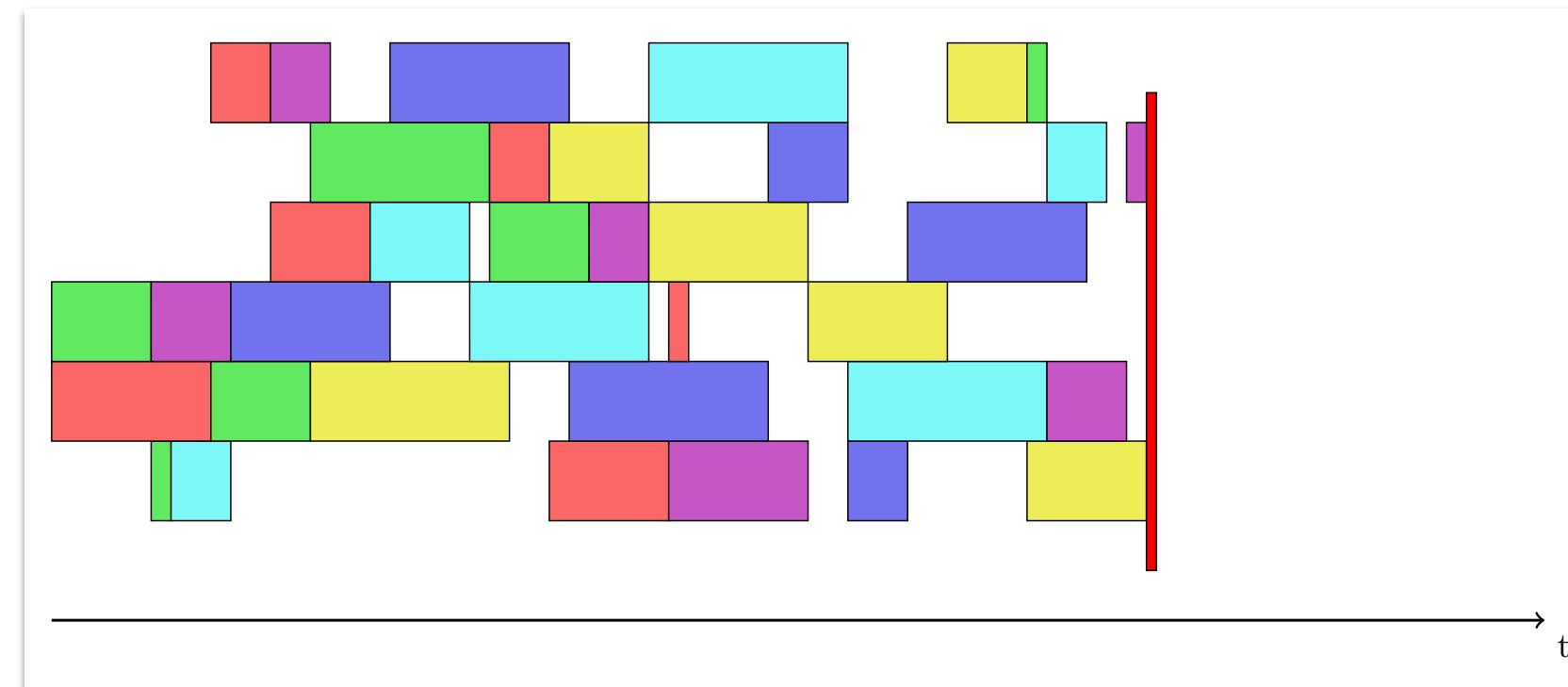
**Only enqueue the constraints with some domain change in their scope (including  $c$  itself)!**

# In next part, design an efficient CP solver

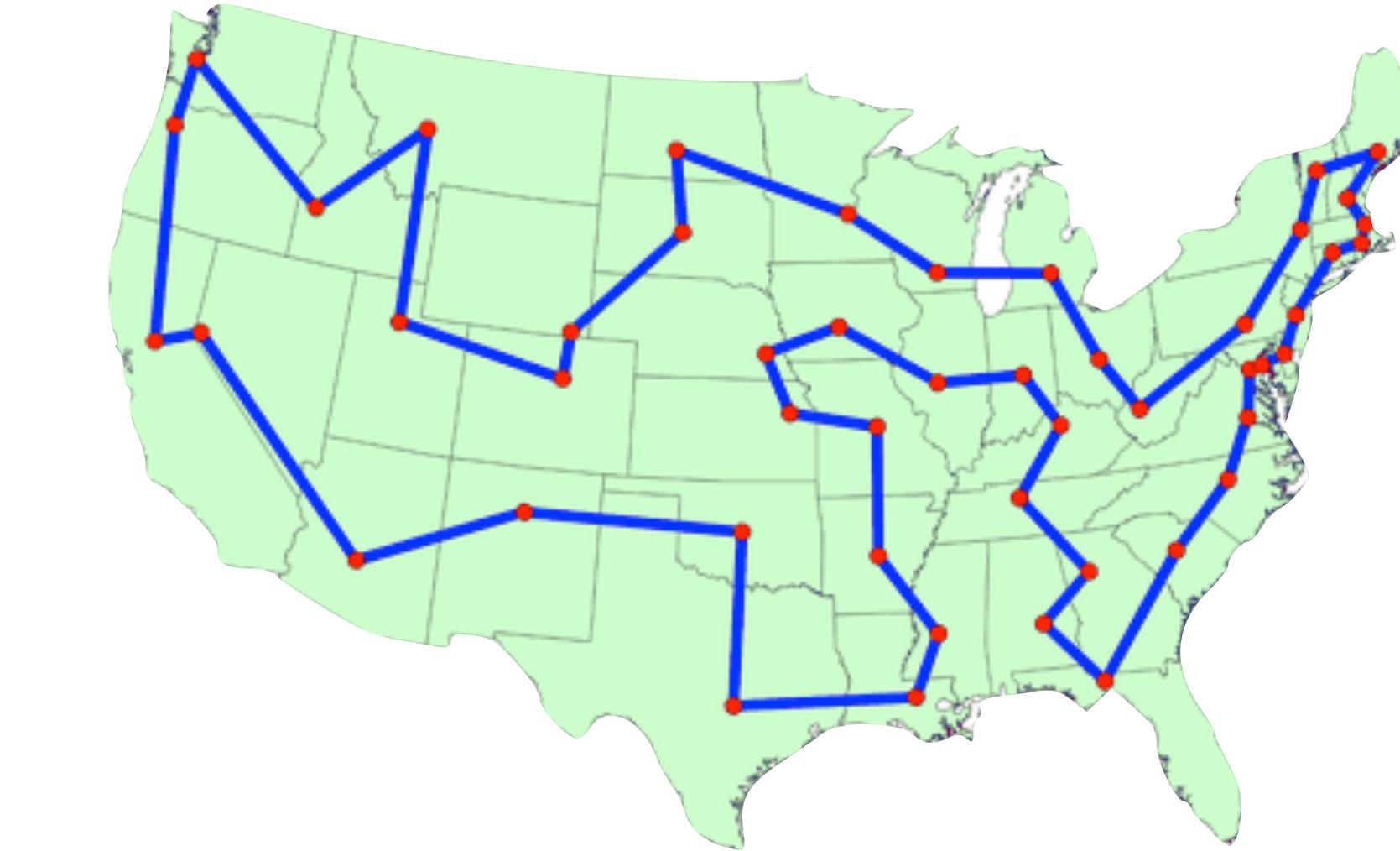
1. More fined grained mechanism for the fix-point and constraint propagation
2. Avoid creating “clones” of the domains and use memory efficient data-structure to restore domains without creating objects
3. Implement a generic and flexible search that can easily be used for complex branching decisions and complex heuristics

# Application Domains

## Scheduling



## Routing



## Rostering

Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon							
6	14	22	6	14	22	6	14	22	6	14	22	6	14	22
Maximum consecutive working days for Ann: 5														
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
A	?	?	A	?	?	A	?	?	A	?	?	A	?	?
1	2	3	4	5	6	7								
Minimum consecutive free days for Beth: 2														
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
?	B	?	?	?	?	B	?	?	?	?	?	?	C	?
1	2												F	
After a night shift sequence: 2 free days      Unwanted pattern: E-L-E														
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
?	?	D	?	?	D	?	?	?	D	?	?	E	?	?
N	N	N			F				E		L	E		

# TSP Modeling: CP vs MIP

MIP

$$\text{minimize} \sum_{i,j} d_{ij} \cdot x_{ij}$$

$$\text{subject to } \sum_{i \in V} x_{ij} = 2 \quad \forall i \in V$$

$$\sum_{i,j \in S, i \neq j} x_{ij} \leq |S| - 1 \quad \forall S \subset V, S \neq \emptyset$$

$$x_{ij} \in \{0,1\}$$

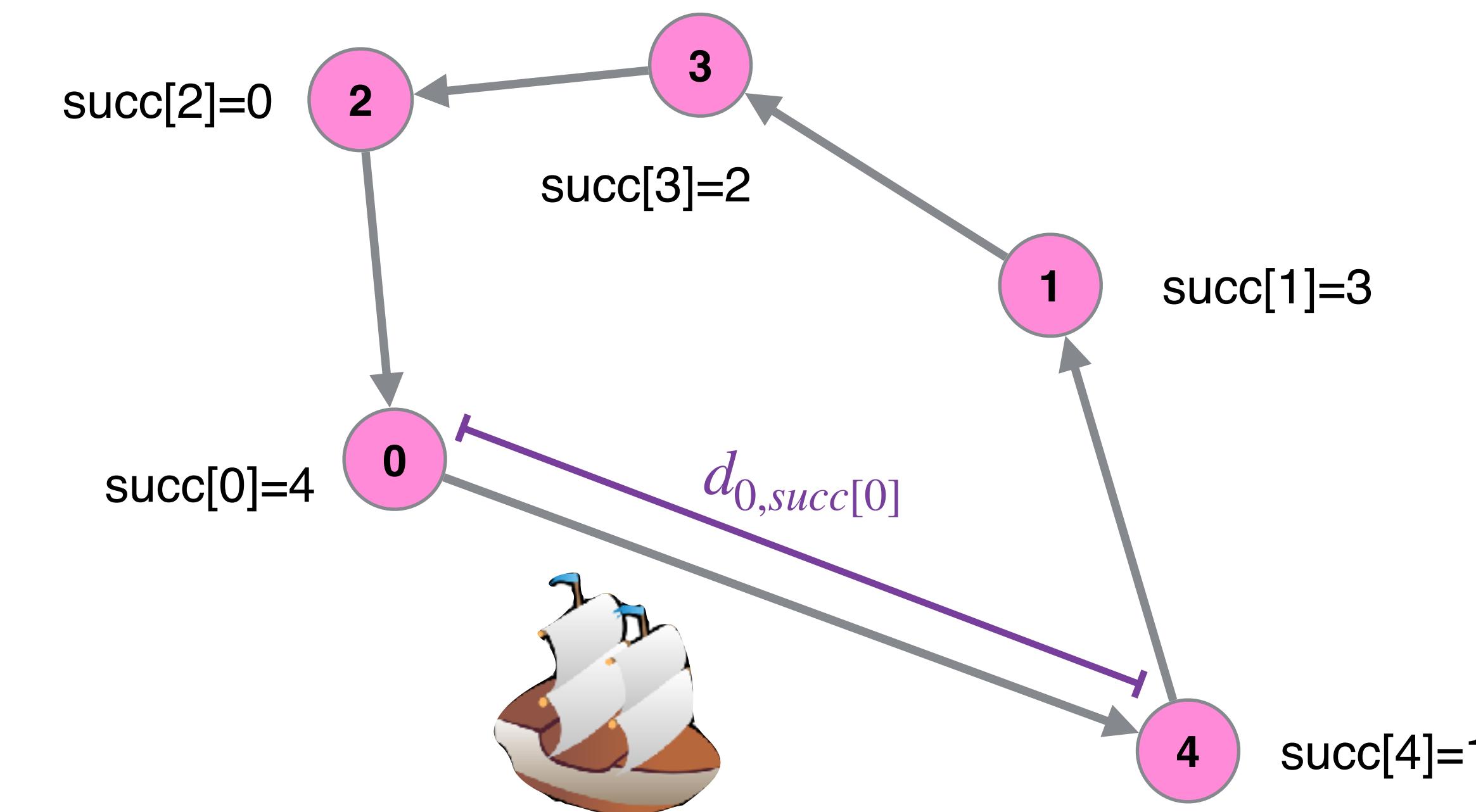
CP

index an array with variables!

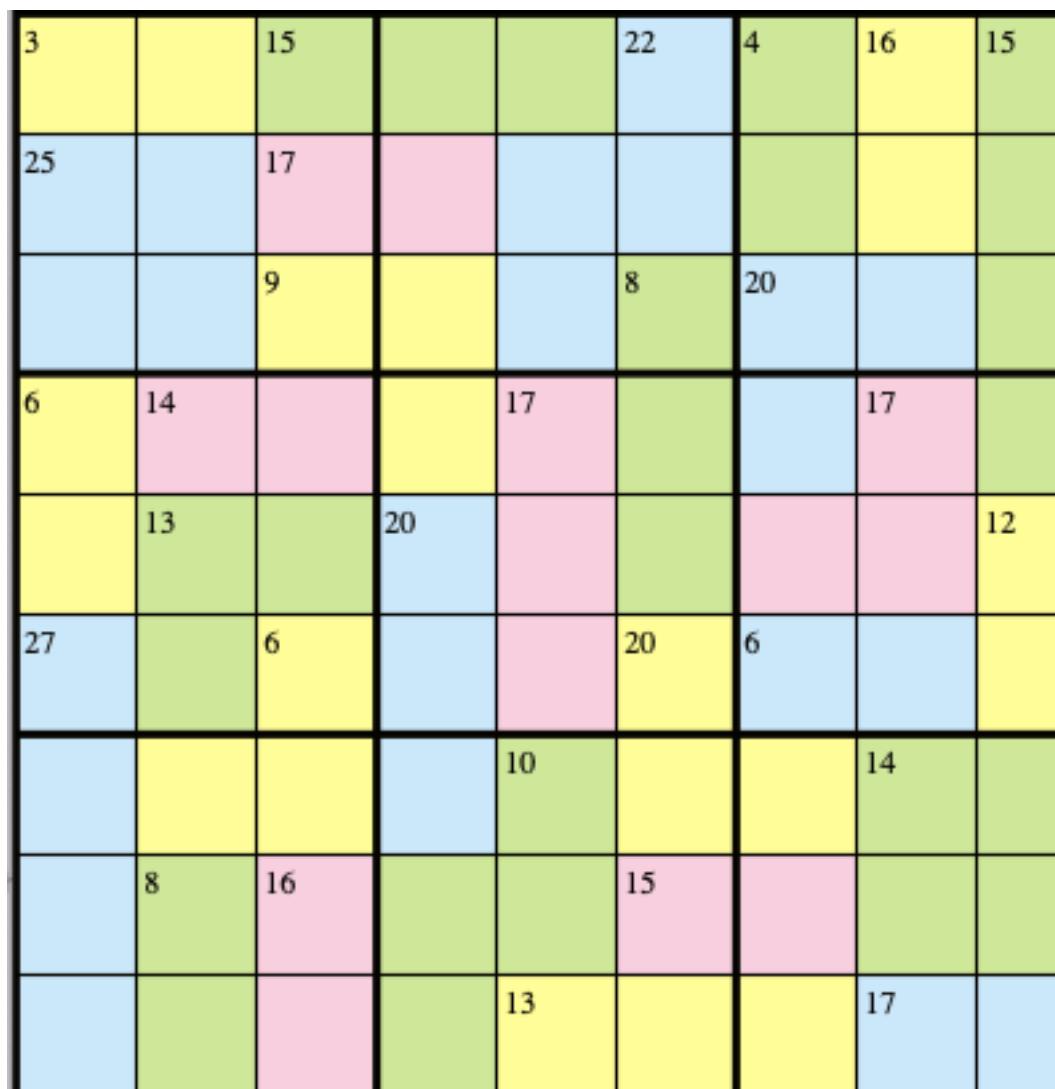
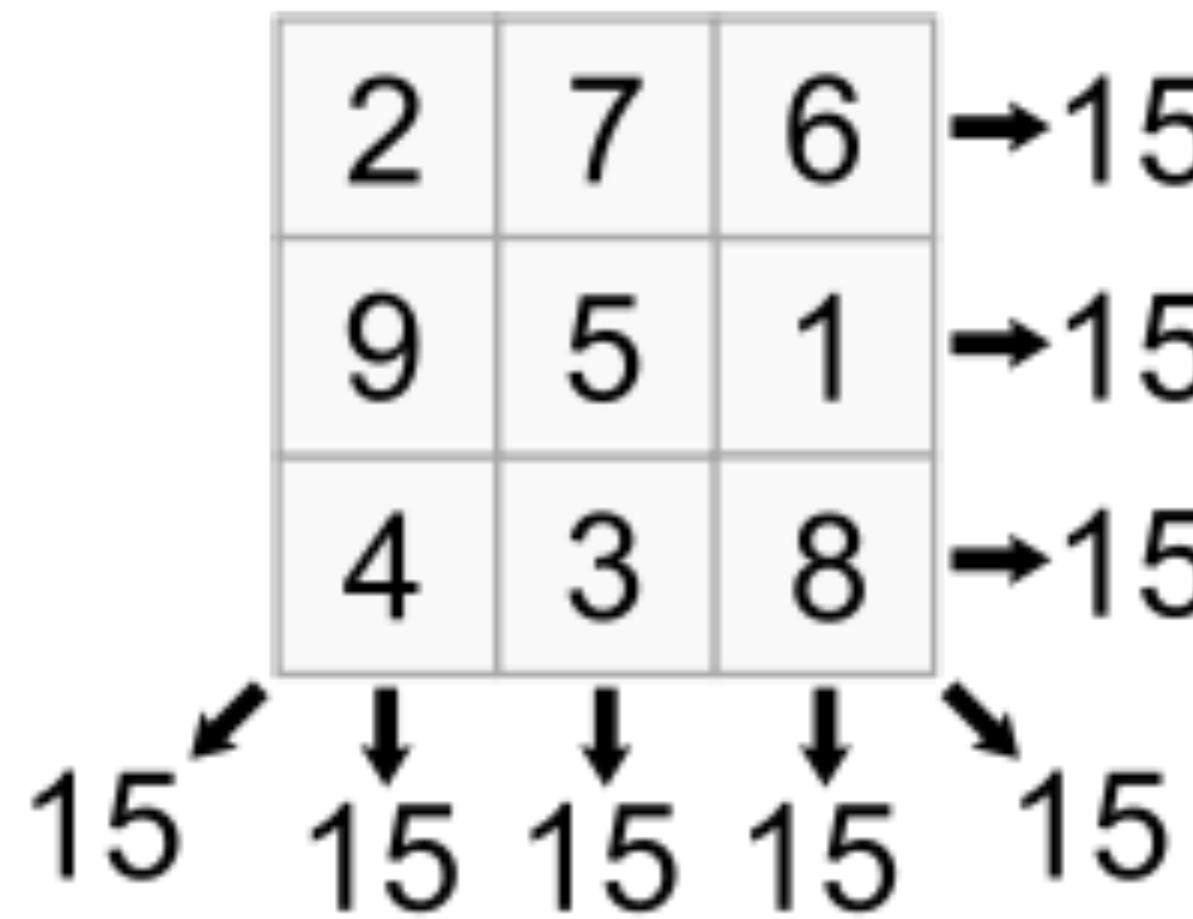
$$\text{minimize} \sum_{i \in V} d_{i,\text{succ}[i]}$$

subject to Circuit(succ)

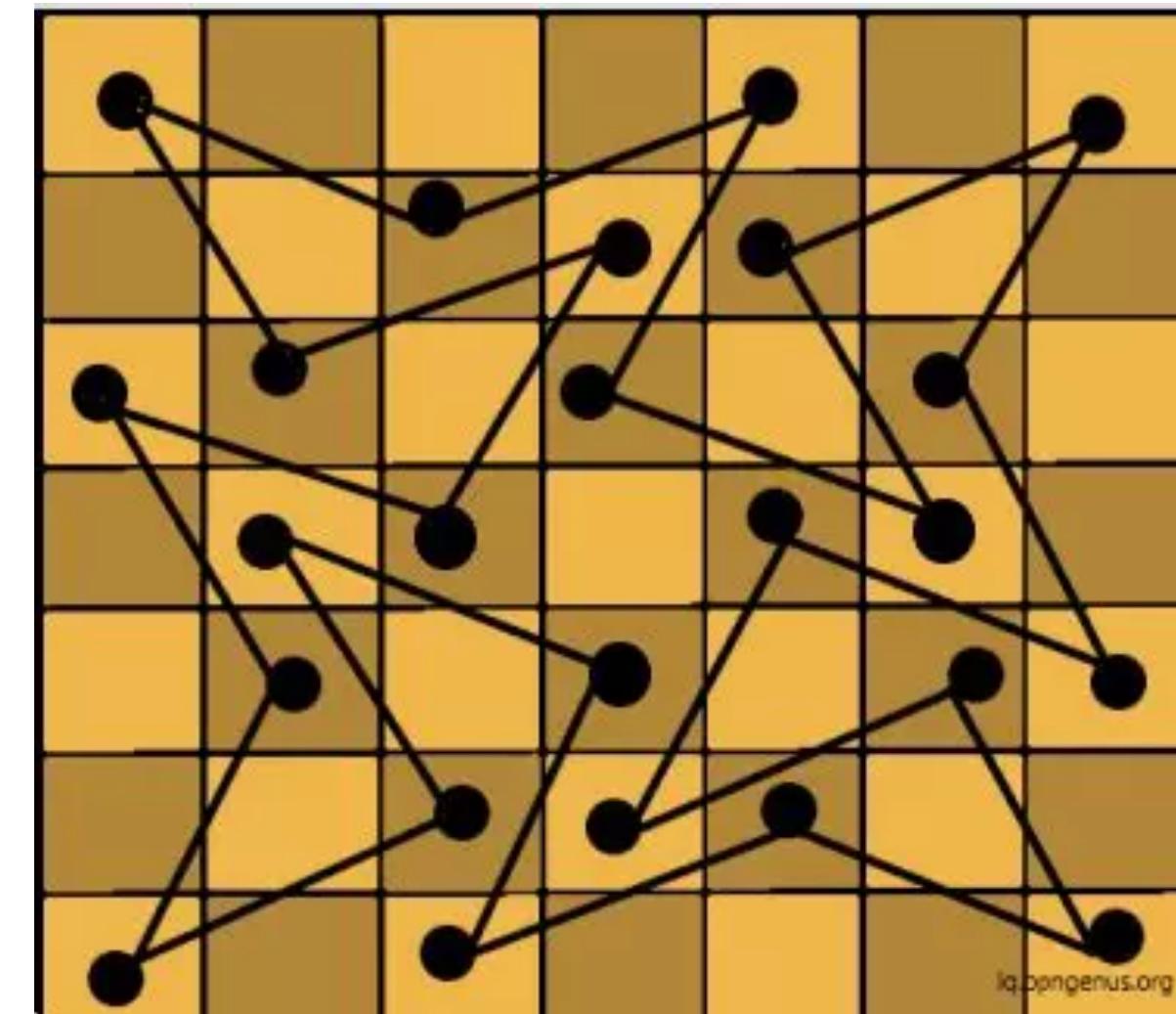
$$\text{succ}[i] \in \{0, \dots, i-1, i+1, n-1\}$$



# Projects Magic Square + Killer Sudoku + KnightTour



3			15			22	4	16	15
25			17						
			9			8	20		
6	14			17				17	
	13		20						12
27		6			20	6			
				10				14	
8		16			15				
				13				17	



# Interested to know more about CP

- ▶ Then follow LINFO2365 (S2), you will
  - Implement a real constraint programming solver called MiniCP
    - You'll see, there are many implementation tricks and algorithms in it.
  - Implement very useful constraints for routing, scheduling, rostering, etc
  - Solve optimization problems (routing, scheduling) (exactly and approximately with CP)
  - Same teaching approach as this one with a final project that uses your solver.