INTRODUCTION À LA PROGRAMMATION PAR CONTRAINTES

Cours Turing+ / Session 1

L’optimisation discrète est partout

- Création d’horaires aux CFF
- Création d’horaires dans les écoles
- Planification des jours de travail dans un hôpital
- Organisation des journées à thèmes au gymnase
- Optimisation de portefeuille financier
- Optimisation de lignes d’assemblage dans l’industrie
- Optimisation d’entrepôts et de la logistique dans les grosses entreprises

En pratique, les problèmes sont compliqués

- Données du problème changent à chaque fois
- Aucune méthode «toute prête» qui fonctionne parfaitement
- Correspond rarement aux «jolis» cas étudiés à l’université
- Chaque problème est unique

Paradigmes de programmation

Programmation impérative

- Programme = Algorithmes + Structures de données

On explique en détail tout ce que le programme doit faire et les données qu'il doit manipuler

➔ Dès que le problème change, il faut pratiquement tout changer

Programmation orientée objets

- Programme = Objets + méthodes
- Tout est objet

On modélise la réalité sous forme d'objets qui interagissent

➔ Lorsque le problème change, les changements sont généralement isolés et facilement identifiables

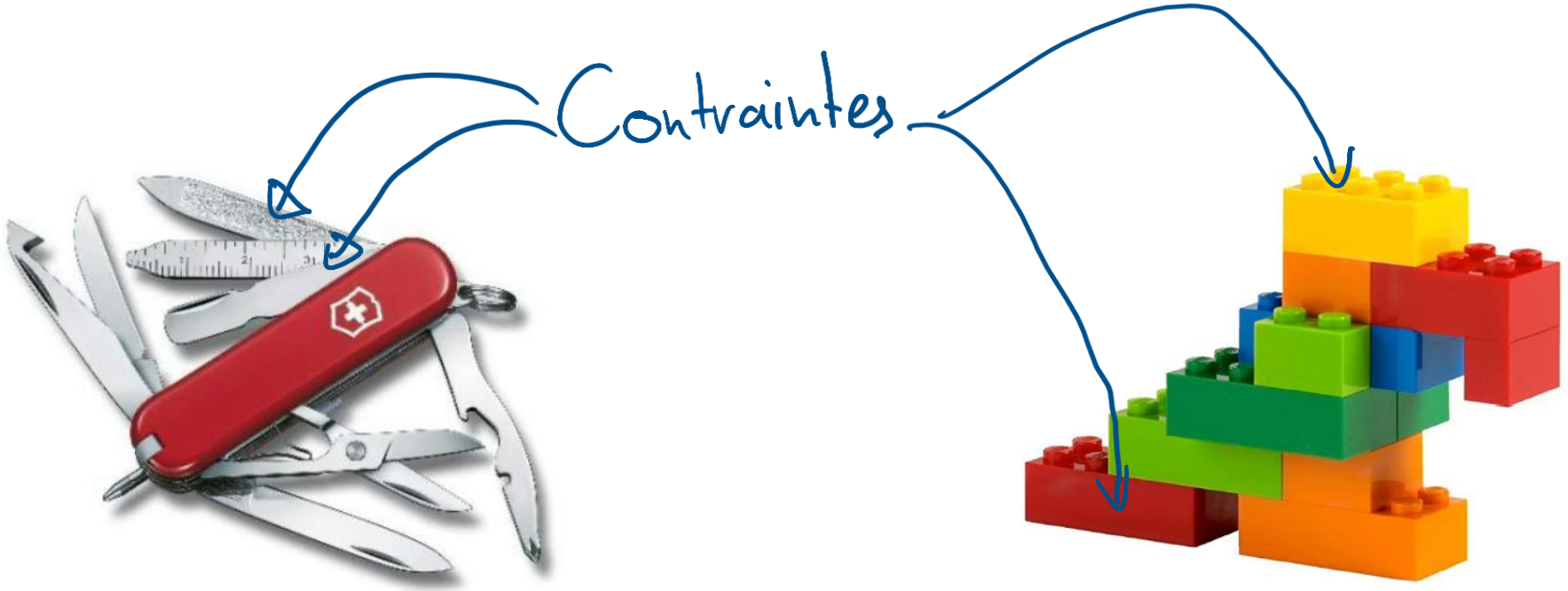
Programmation par contraintes

- CP = Modèle + recherche
- L'utilisateur formule le problème (modèle)
- L'ordinateur (Solveur de contraintes) recherche les solutions

On décrit le problème sous forme de variables et de contraintes

➔ On décrit la logique du problème sans spécifier en détail comment résoudre le problème

Programmation par contraintes



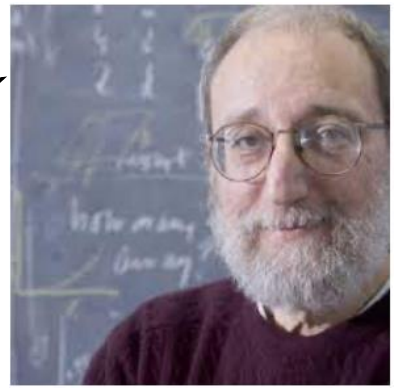
Les outils du couteau suisse / les blocs DUPLO représentent les contraintes permettant de décrire le problème

Programmation par contraintes

"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)

langage de modélisation de haut niveau

```
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);
   }
}
```



E. Freuder

Objectifs d’apprentissage

1. Construire le couteau suisse (solveur de contraintes)
2. Utiliser le couteau suisse (pour résoudre des problèmes complexes)

Plan

Problèmes de satisfaction de contraintes

- Problème des n-dames

Trois approches

- Naïve : générer et tester (DFS + filtrage)
- DFS + élagage
- Approche générique

Révisions concernant la récursion

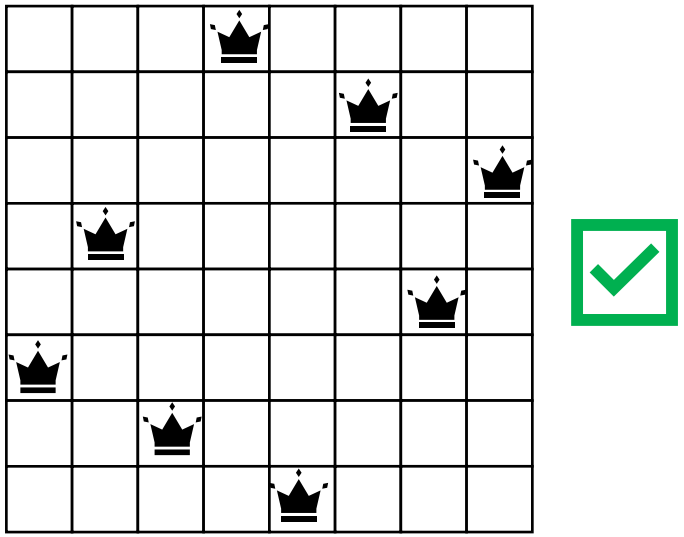
- Première approche avec la POO
- Framework générique et réutilisable avec les variables, domaines et contraintes
- Première approche avec les contraintes et les domaines

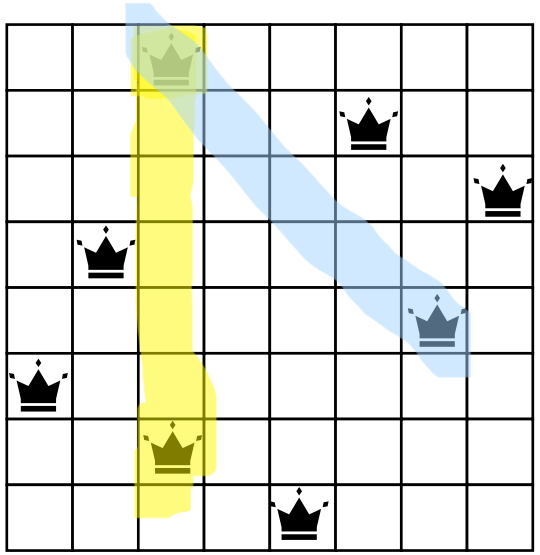
Problème des N dames

Description : Placer n (disons 8) reines sur un échiquier $n \times n$ sans qu'elles ne s'attaquent mutuellement

Il ne doit donc pas y avoir plus d'une reine par

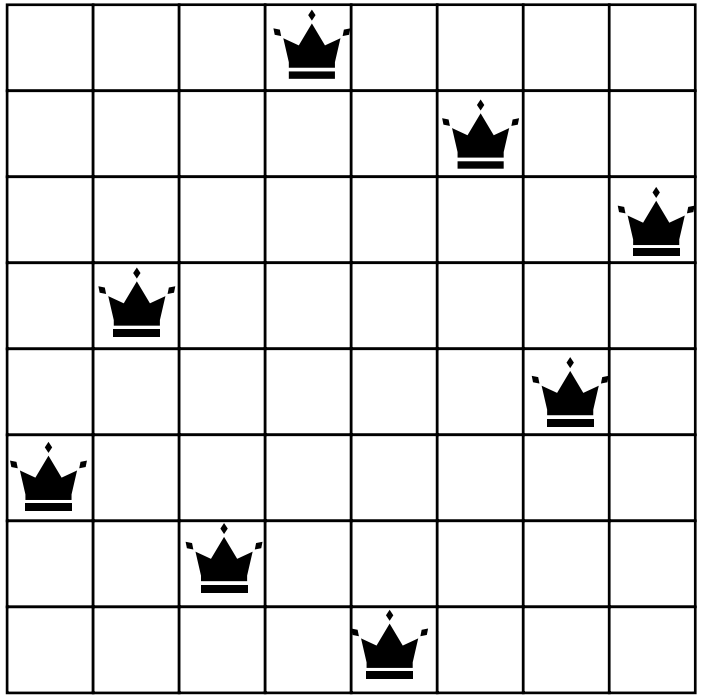
- Colonne
- Ligne
- Diagonale





Modélisations possibles : n^2 variables 0/1

- Tenir compte du fait qu'il n'y a de toute manière qu'une seule reine par colonne
- Déterminer la ligne dans laquelle placer chaque reine



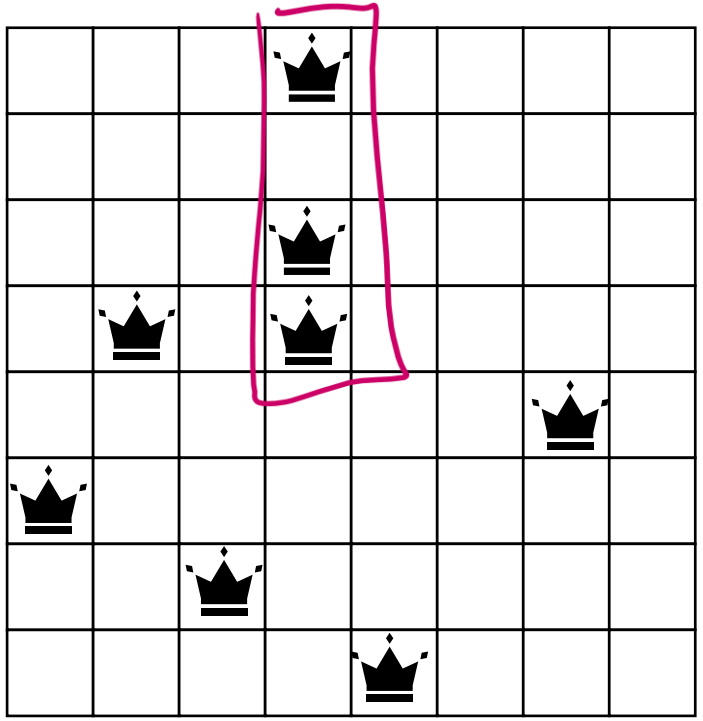
0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0

Désavantage

- n^2 variables
- Il faut vérifier les contraintes de colonnes, lignes et diagonales

Modélisations possibles : n^2 variables 0/1

- Tenir compte du fait qu'il n'y a de toute manière qu'une seule reine par colonne
- Déterminer la ligne dans laquelle placer chaque reine



0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0

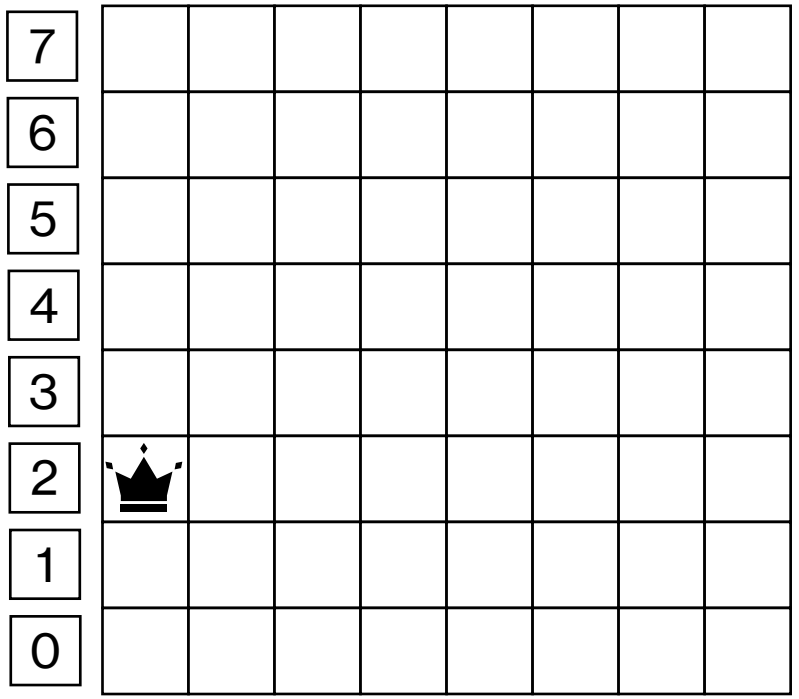
Désavantage

- n^2 variables
- Il faut vérifier les contraintes de colonnes, lignes et diagonales

Configuration possible ⇒ il faut vérifier les contraintes de colonnes

Modélisations possibles : n variables $0..n-1$

Représenter l'échiquier comme $n = 8$ variables booléennes $x_i \in D = \{0, 1, 2, 3, 4, 5, 6, 7\}$



Variables de décision

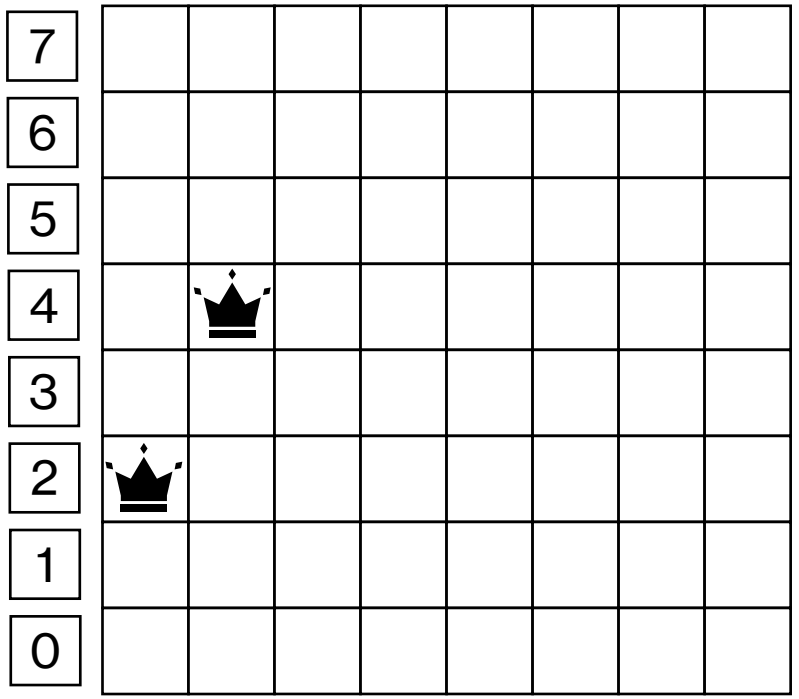
2	?	?	?	?	?	?	?
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Avantage

- n variables
- Pas besoin de vérifier les contraintes de colonne

Modélisations possibles : n variables $0..n-1$

Représenter l'échiquier comme $n = 8$ variables booléennes $x_i \in D = \{0, 1, 2, 3, 4, 5, 6, 7\}$



Variables de décision

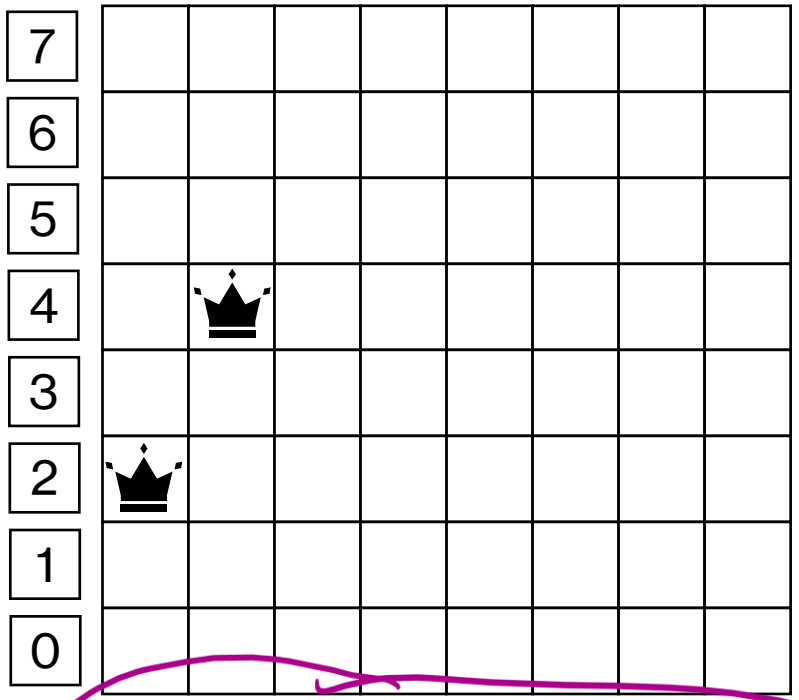
2	4	?	?	?	?	?	?
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Avantage

- n variables
- Pas besoin de vérifier les contraintes de colonne

Modélisations possibles : n variables $0..n-1$

Représenter l'échiquier comme $n = 8$ variables booléennes $x_i \in D = \{0, 1, 2, 3, 4, 5, 6, 7\}$



Variables de décision

2	4	?	?	?	?	?	?
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

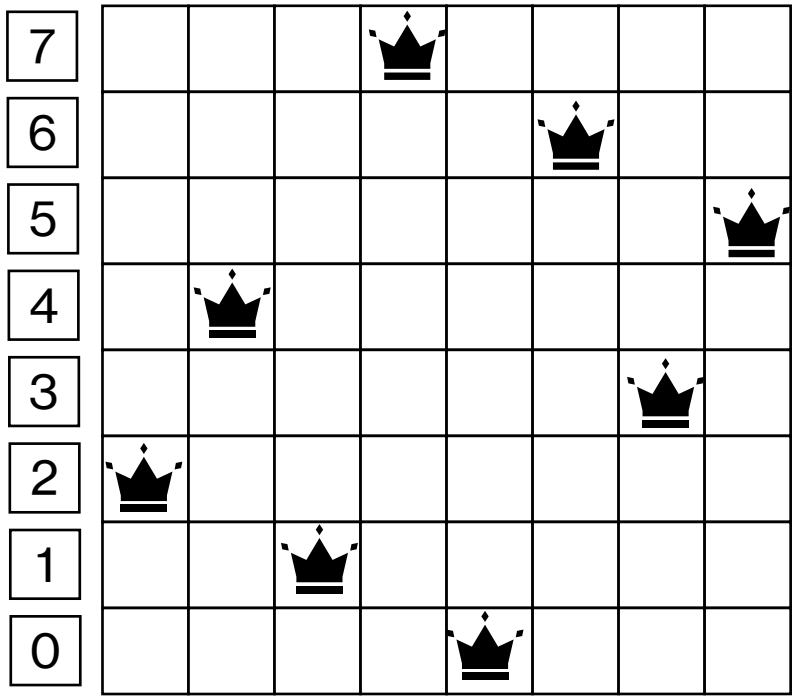
Avantage

- n variables
- Pas besoin de vérifier les contraintes de colonne

Résoudre à la main

Modélisations possibles : n variables $0..n-1$

Représenter l'échiquier comme $n = 8$ variables booléennes $x_i \in D = \{0, 1, 2, 3, 4, 5, 6, 7\}$



Variables de décision

2	4	1	7	0	6	3	5
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Avantage

- n variables
- Pas besoin de vérifier les contraintes de colonne

Stratégie 1 : générer et tester

1. Générer toutes les configurations possibles ($n^n = 8^8 = 2^{24} \approx 16 \times 10^6$)
2. Garder les solutions qui respectent toutes les contraintes

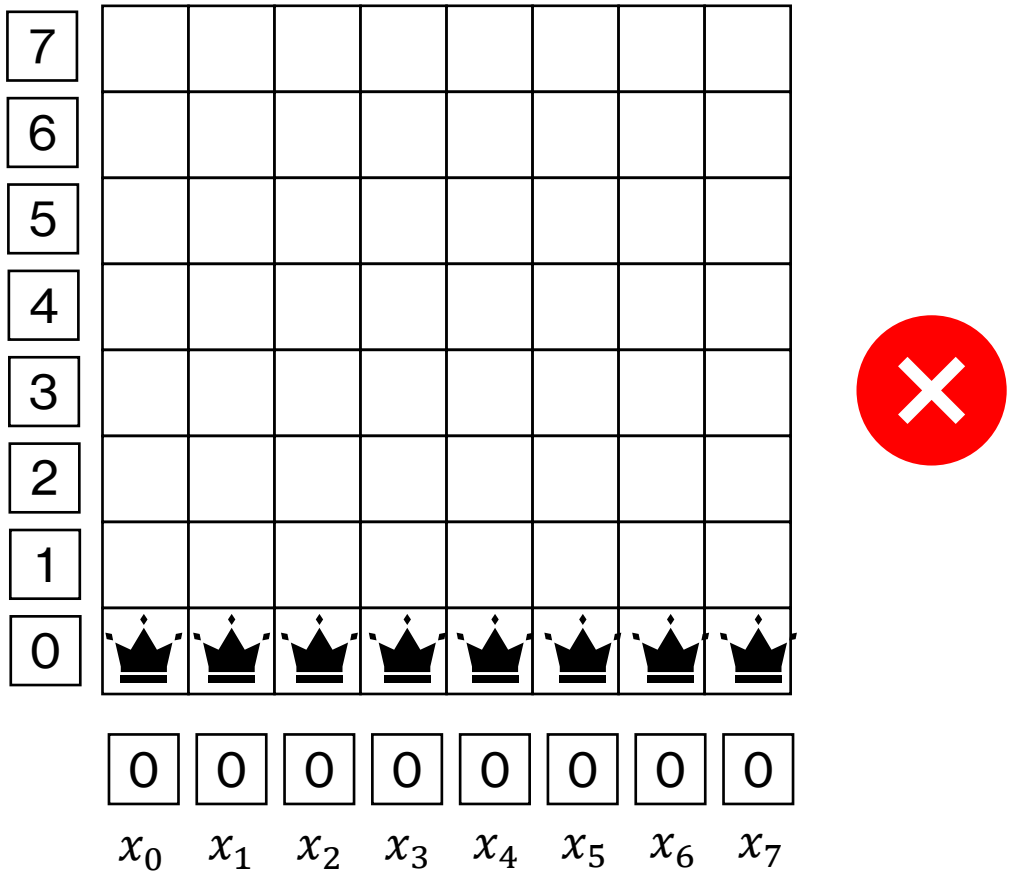
1 Générer toutes les «solutions» (recherche systématique)

→

2 Garder les **solutions faisables** (qui satisfont toutes les contraintes)

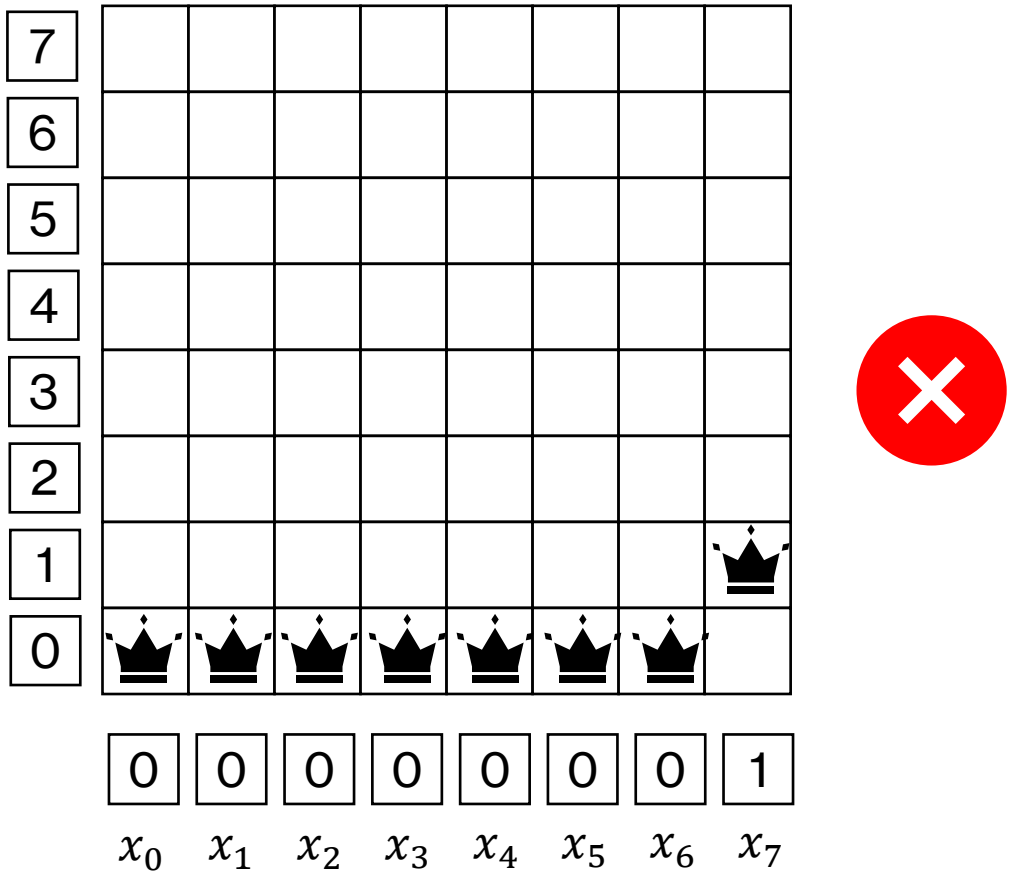
Stratégie 1 : générer et tester

1. Générer toutes les configurations possibles ($n^n = 8^8 = 2^{24} \approx 16 \times 10^6$)
2. Garder les solutions qui respectent toutes les contraintes



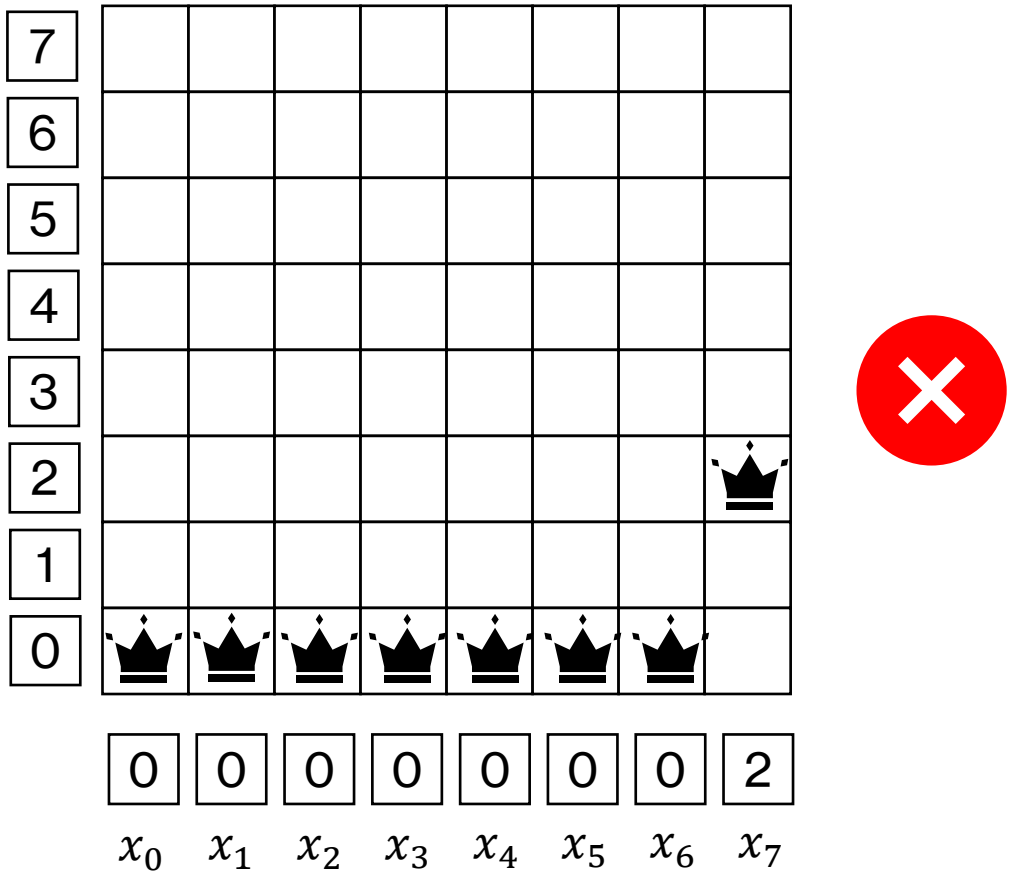
Stratégie 1 : générer et tester

1. Générer toutes les configurations possibles ($n^n = 8^8 = 2^{24} \approx 16 \times 10^6$)
2. Garder les solutions qui respectent toutes les contraintes



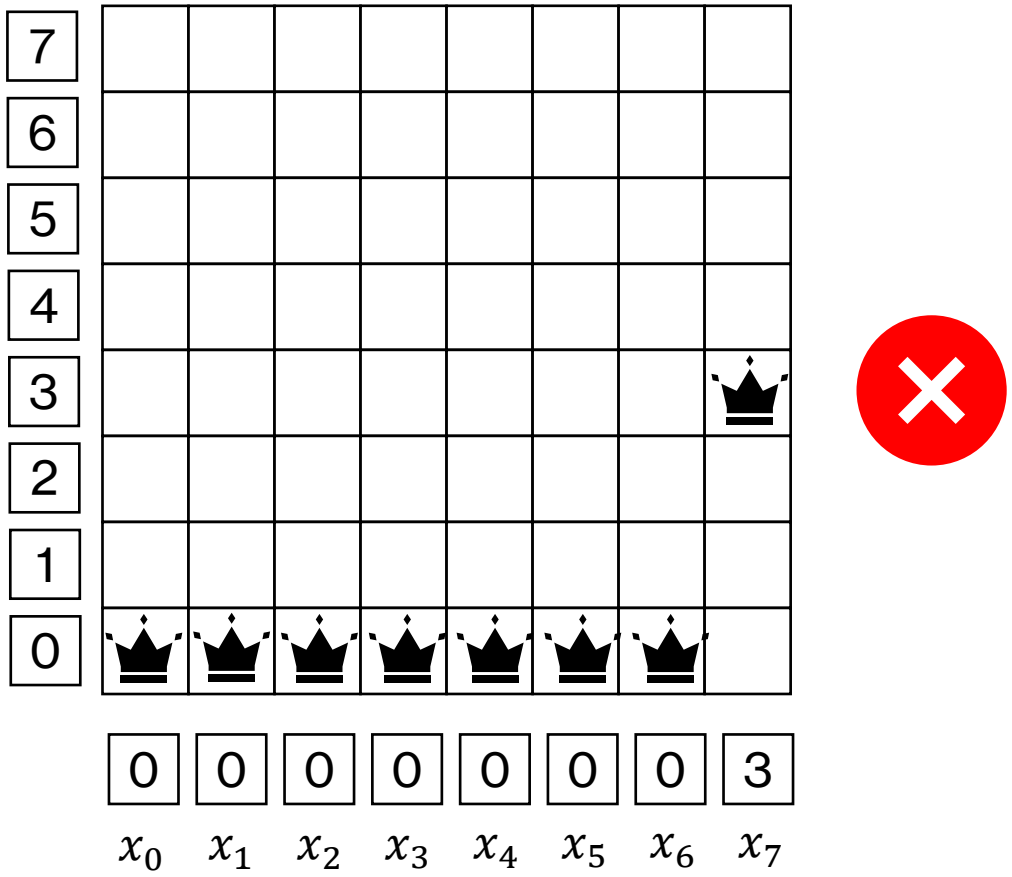
Stratégie 1 : générer et tester

1. Générer toutes les configurations possibles ($n^n = 8^8 = 2^{24} \approx 16 \times 10^6$)
2. Garder les solutions qui respectent toutes les contraintes



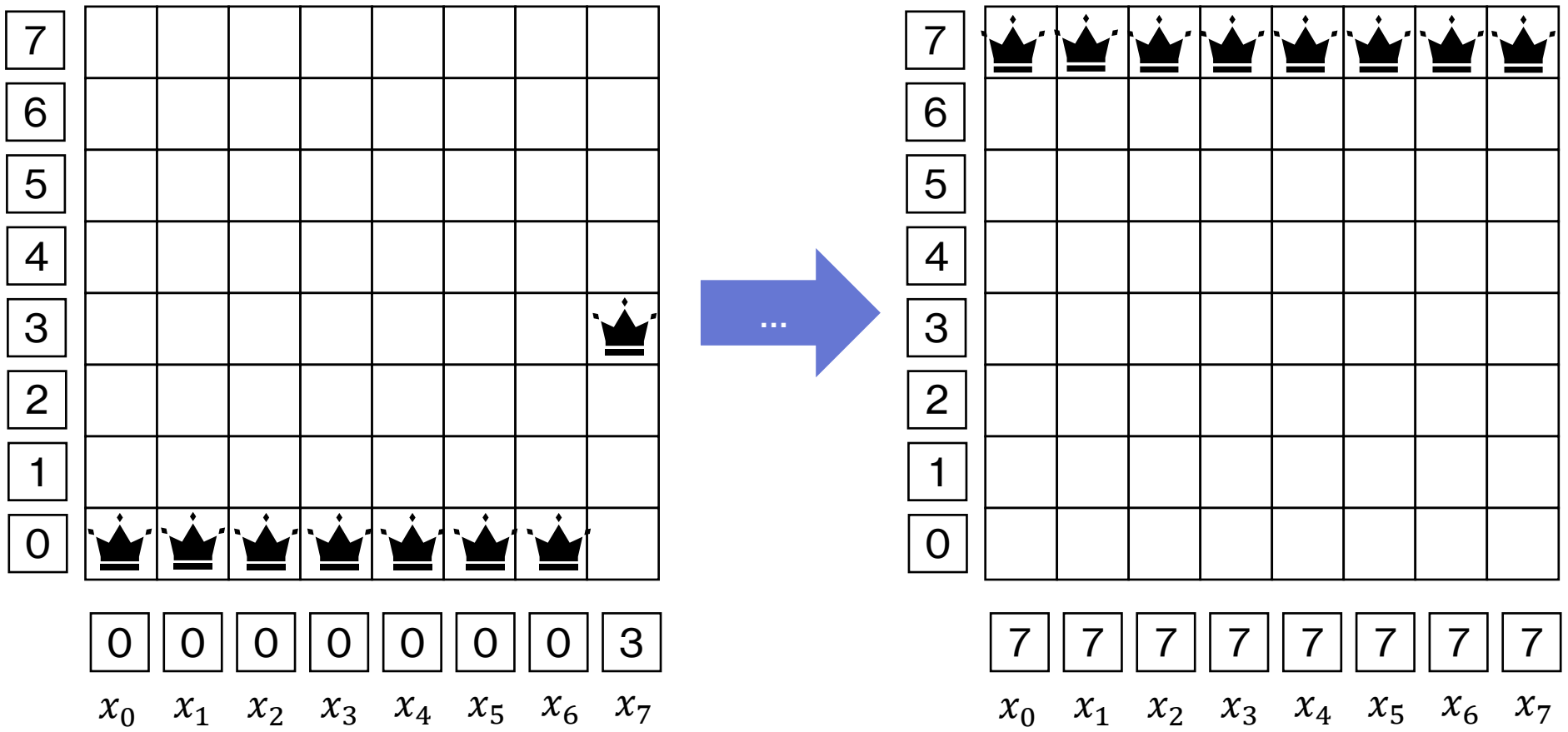
Stratégie 1 : générer et tester

1. Générer toutes les configurations possibles ($n^n = 8^8 = 2^{24} \approx 16 \times 10^6$)
2. Garder les solutions qui respectent toutes les contraintes



Stratégie 1 : générer et tester

1. Générer toutes les configurations possibles ($n^n = 8^8 = 2^{24} \approx 16 \times 10^6$)
2. Garder les solutions qui respectent toutes les contraintes

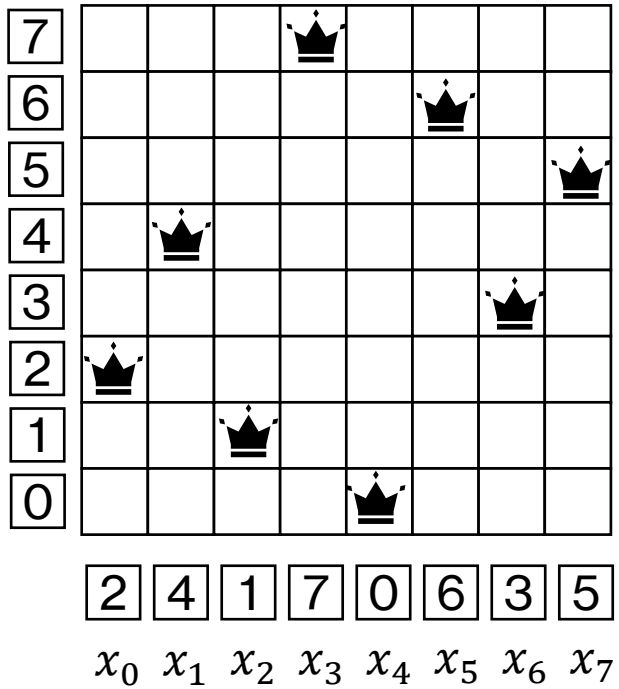


Stratégie 1 : générer et tester

- La première représentation possède 2^{64} configurations possibles
- La deuxième représentation ne possède «que» $n^n = 8^8 = 2^{24} \approx 16$ millions de configurations

0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0

- $n^2 = 64$ variables
- 2^{64} configurations
- Il faut vérifier les contraintes de colonnes



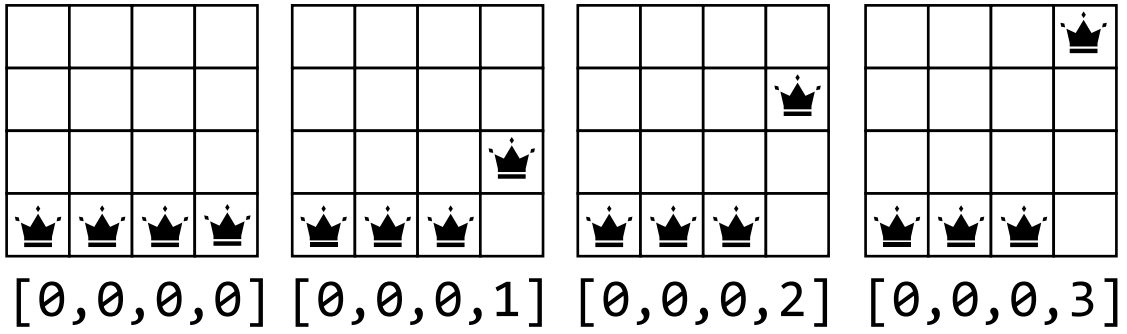
- $n = 8$ variables
- 2^{24} configurations
- Les contraintes de colonne sont implicites dans la modélisation

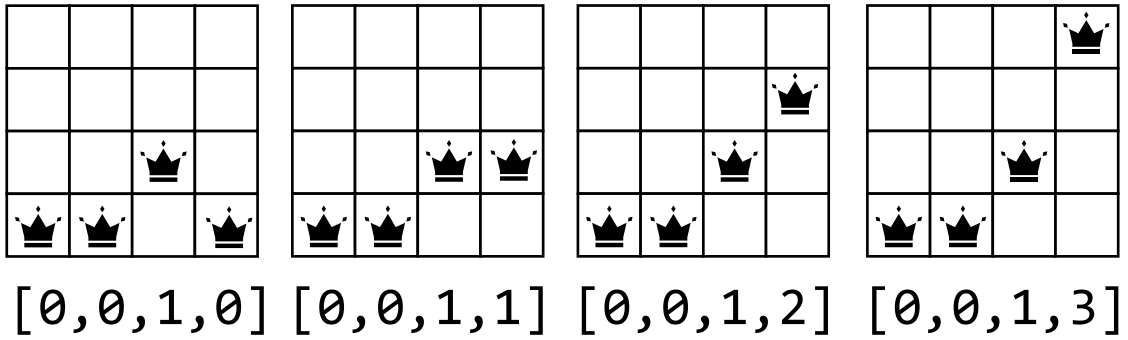
Stratégie 1 : générer et tester

1

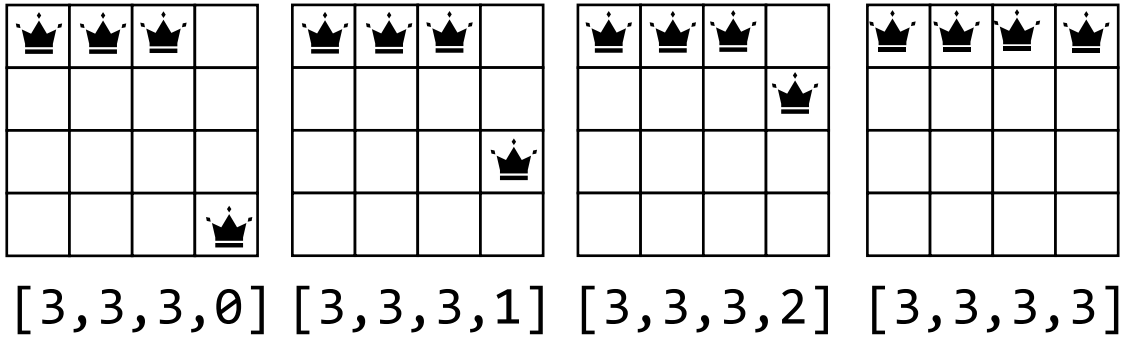
Générer toutes les «solutions»
(recherche systématique)

```
solutions = [
    [0, 0, 0, 0],
    [0, 0, 0, 1],
    [0, 0, 0, 2],
    [0, 0, 0, 3],
    [0, 0, 1, 0],
    [0, 0, 1, 1],
    [0, 0, 1, 2],
    [0, 0, 1, 3],
    ...
    [3, 3, 3, 0],
    [3, 3, 3, 1],
    [3, 3, 3, 2],
    [3, 3, 3, 3],
]
```





...

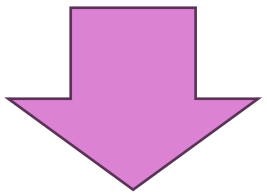


Stratégie 1 : générer et tester

1

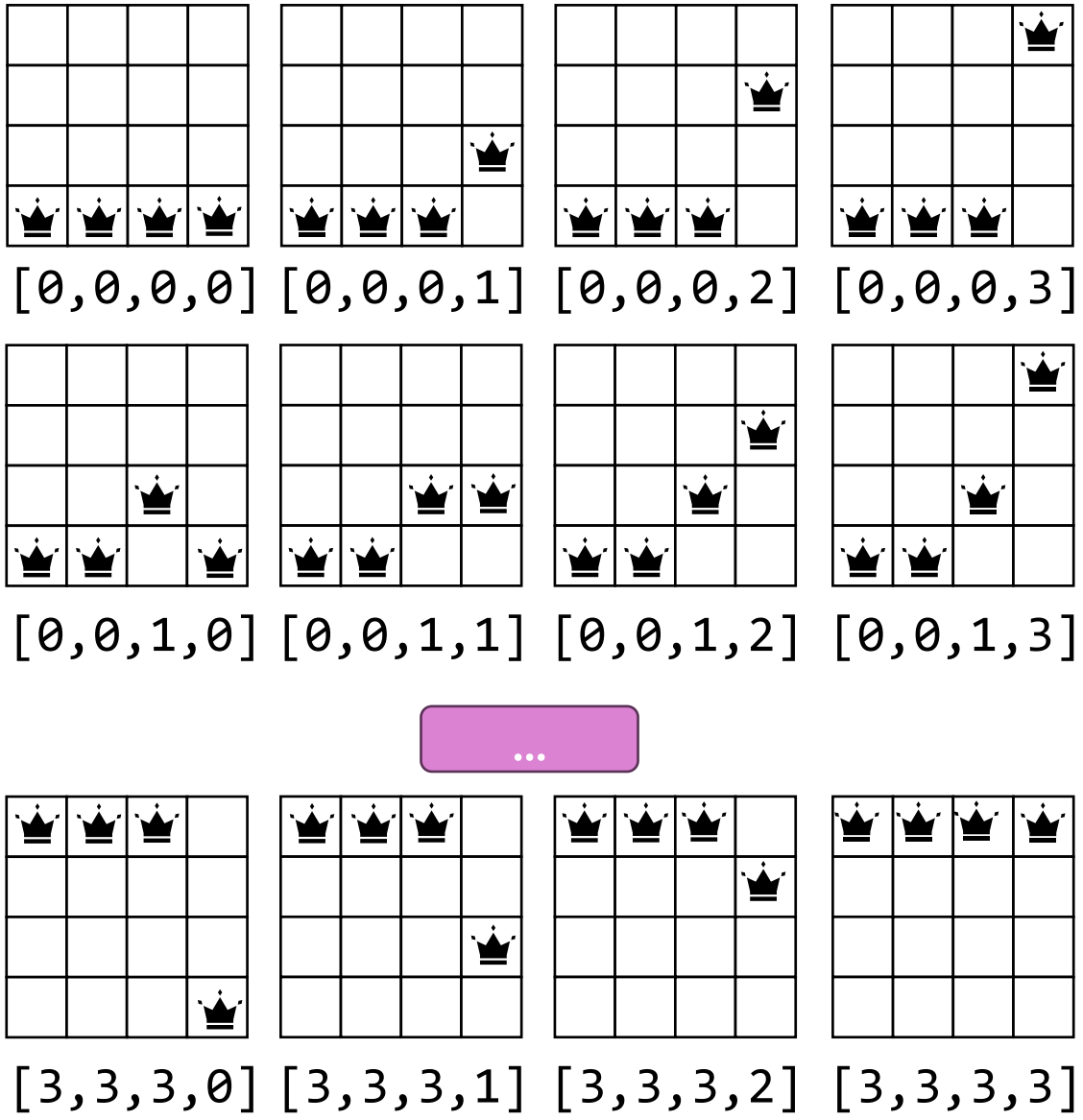
Générer toutes les «solutions»
(recherche systématique)

```
n = 4
solutions = [
    [a,b,c,d] for a in range(n)
        for b in range(n)
        for c in range(n)
        for d in range(n)
]
```



Problème

Il faut modifier le code à chaque fois qu'on change `n`



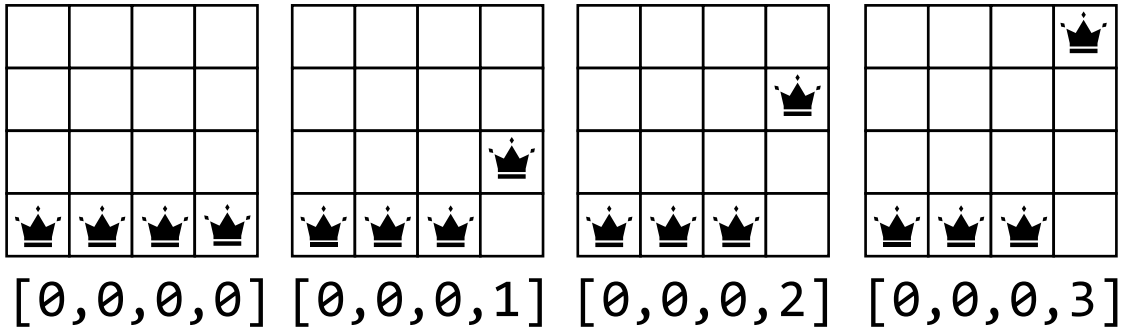
Stratégie 1 : générer et tester

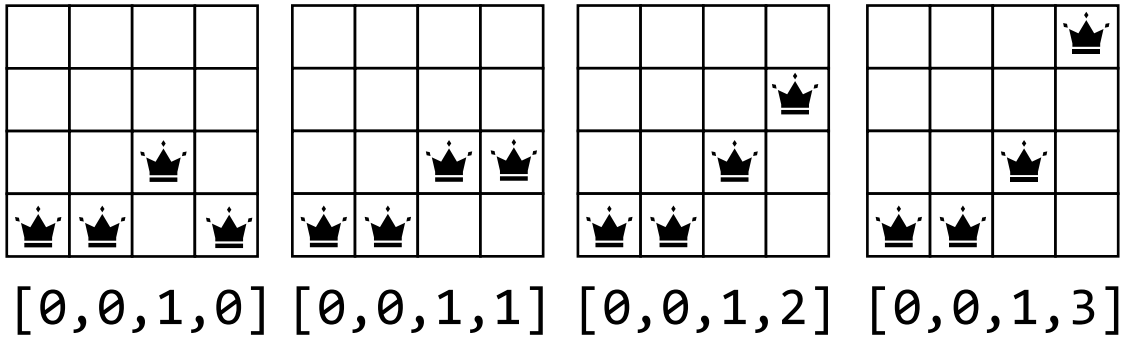
1

Générer toutes les «solutions»
(recherche systématique)

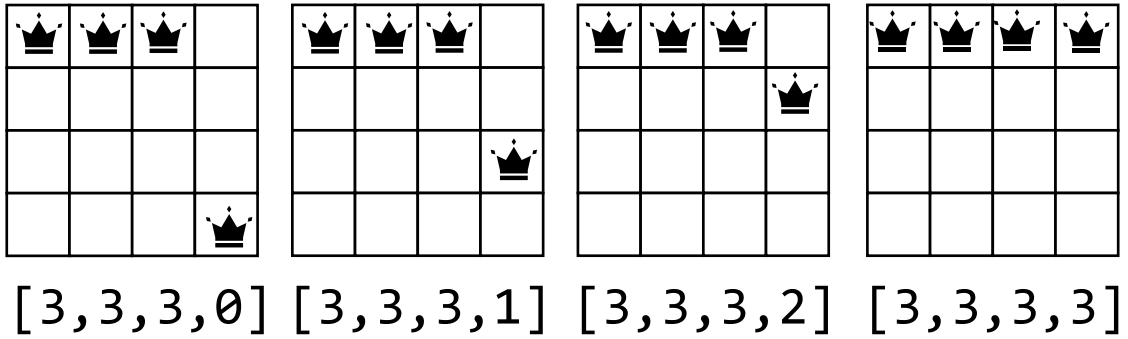
Mieux

```
from itertools import product

def generate_solutions(n):
    dom = list(range(n))
    return list(product(dom, repeat=n))

solutions = generate_solutions(n=4)
```





...

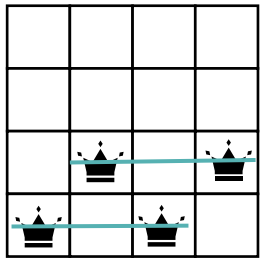


Stratégie 1 : générer et tester

2

Filtrer les **solutions faisables**
(qui satisfont toutes les contraintes)

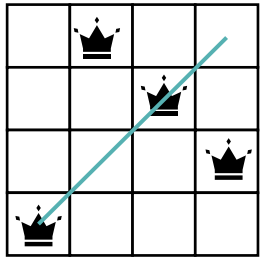
Contraintes de colonnes



`Q=[0,1,0,1]`

2 reines i et j sont sur la même ligne ssi
`Q[i] = Q[j]`

Diagonales montantes



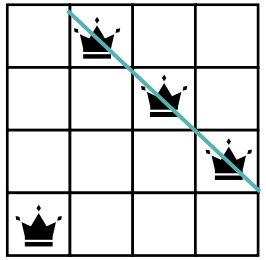
`Q=[0,3,2,1]`

2 reines i et j sont sur la même diagonale / ssi
`Q[i] - Q[j] = i – j`

`Q[0] - Q[2] = 0 - 2`

`  0  -   2  = 0 - 2`

Diagonales descendantes



`Q=[0,3,2,1]`

2 reines i et j sont sur la même diagonale \ ssi
`Q[j] - Q[i] = i – j`

`Q[1] - Q[3] = 3 - 1`

`  3  -   1  = 3 - 1`

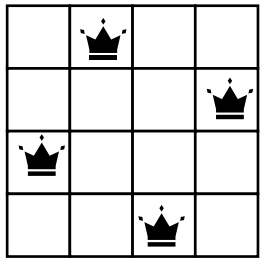
Stratégie 1 : générer et tester

2

Cas de tests pour la fonction `check_constraints`

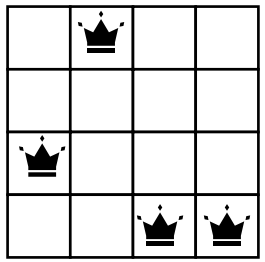
 `Q=[0]`

Toutes les contraintes sont satisfaites pour l'échiquier $n = 1$



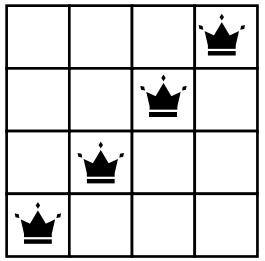
`Q=[1,3,0,2]`

Toutes les contraintes sont satisfaites



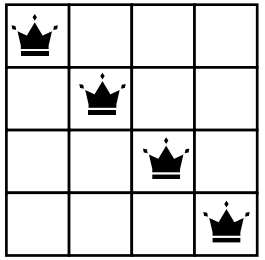
`Q=[1,3,0,0]`

Toutes les contraintes sont satisfaites sauf contrainte de ligne



`Q=[0,1,2,3]`

Toutes les contraintes sont satisfaites sauf les diagonales montantes



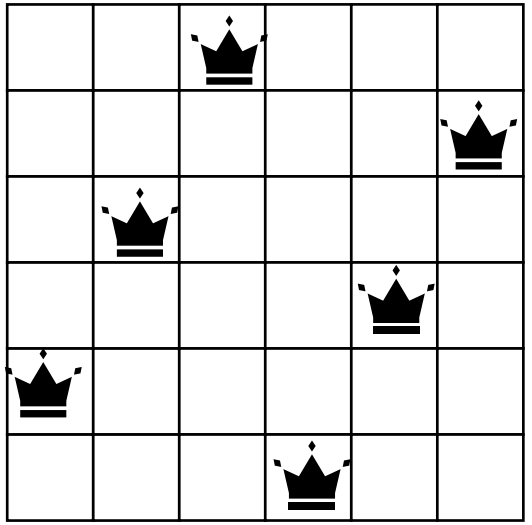
`Q=[3,2,1,0]`

Toutes les contraintes sont satisfaites sauf les diagonales descendantes

Stratégie 1 : générer et tester

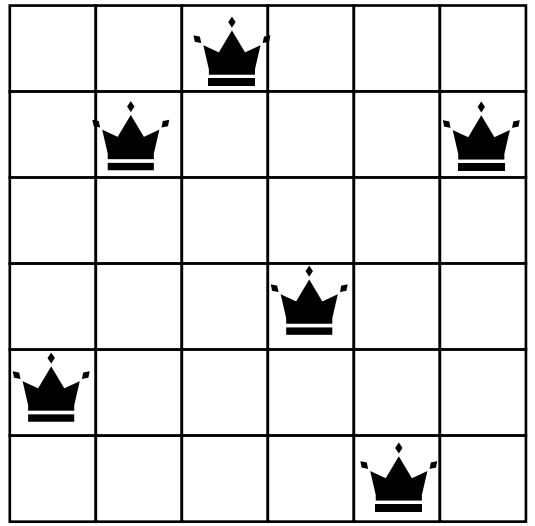
2

Cas de tests pour la fonction `check_constraints`



`[1,3,5,0,2,4]`

Toutes les contraintes sont satisfaites pour une autre taille que $n = 4$



`[1,4,5,2,0,4]`

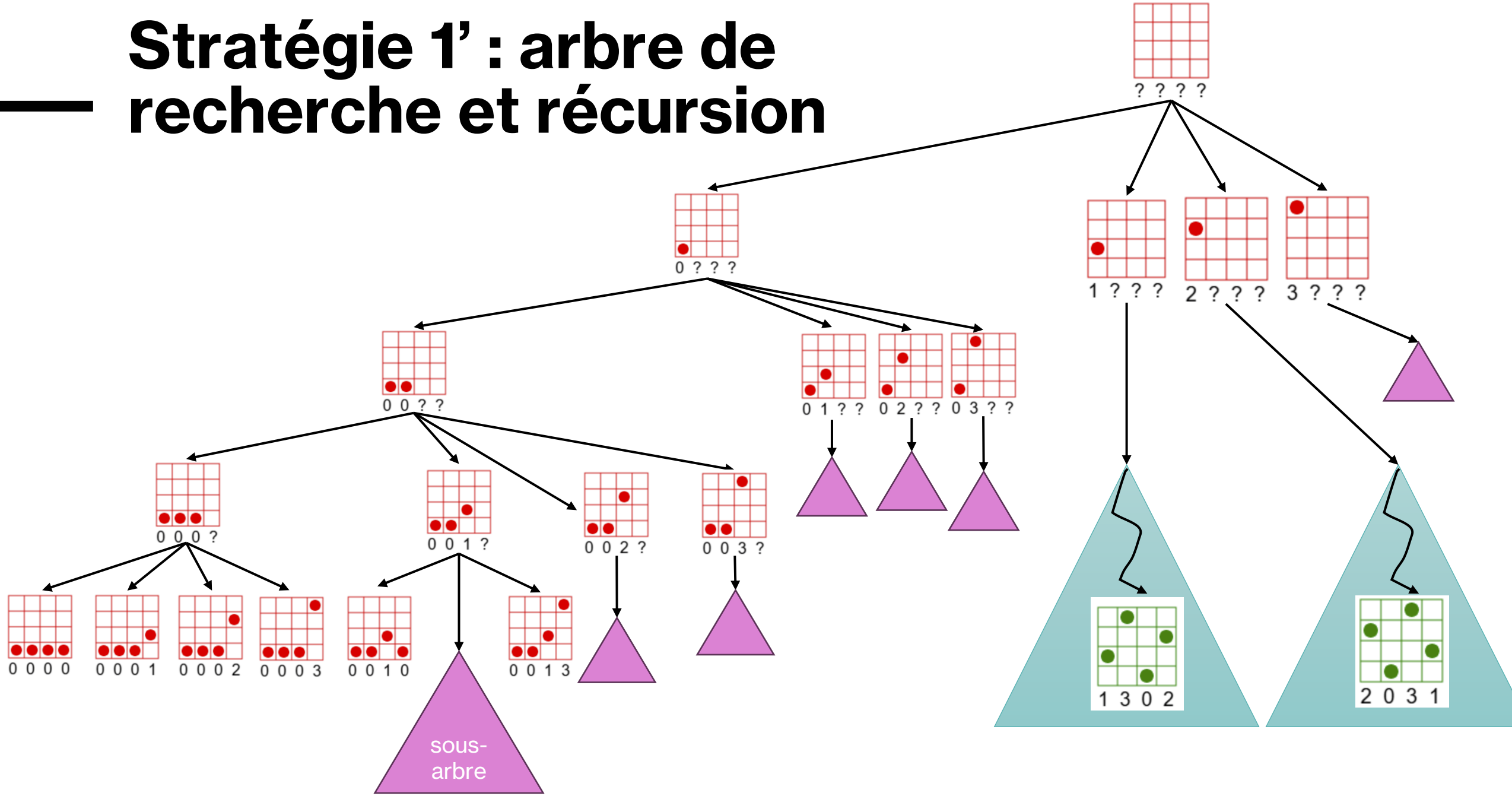
Aucune contrainte n'est satisfaite pour une autre taille que $n = 4$

Stratégie 1 : générer et tester

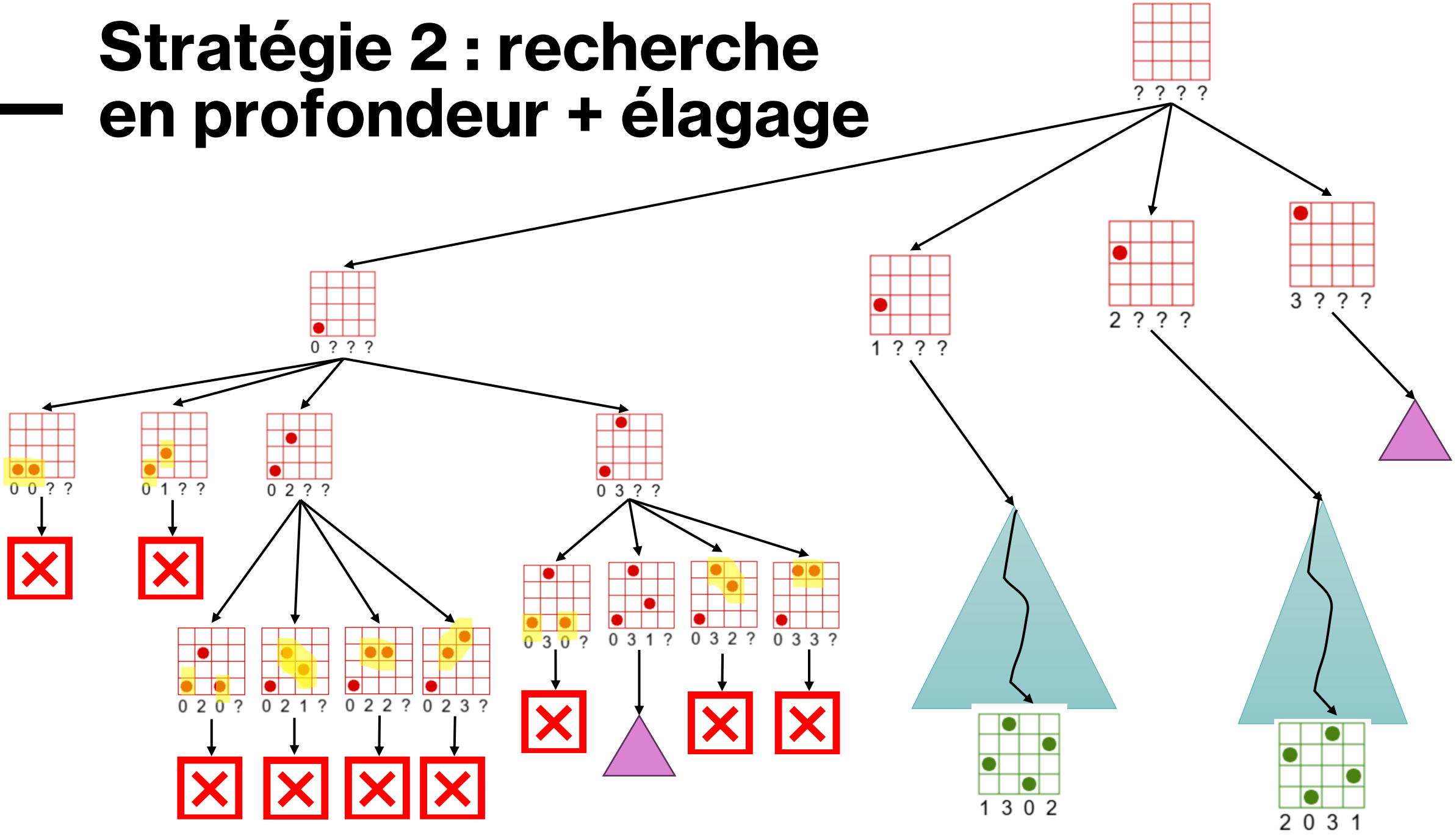
3

Implémentez la solution en Python

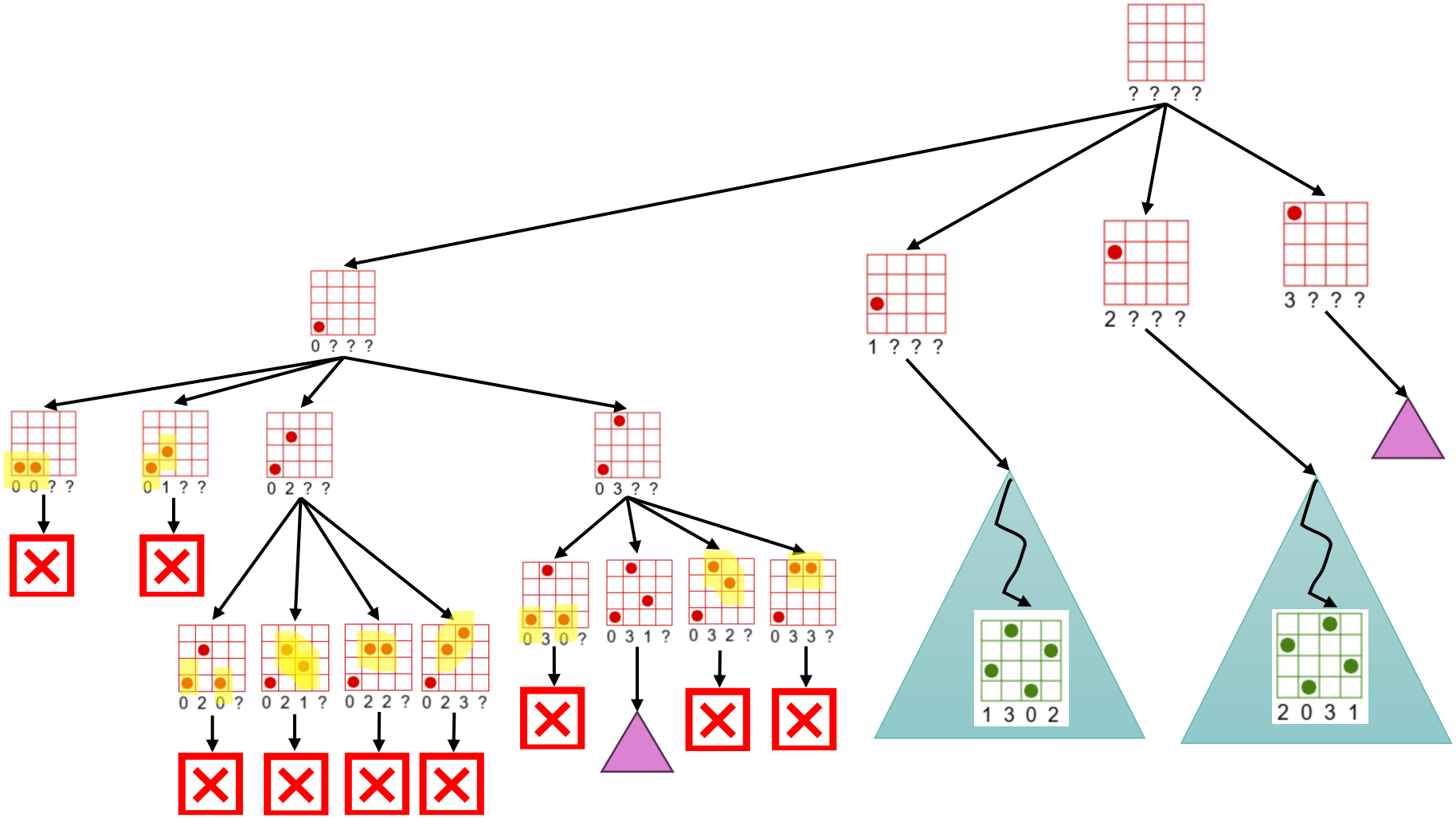
Stratégie 1’ : arbre de recherche et récursion



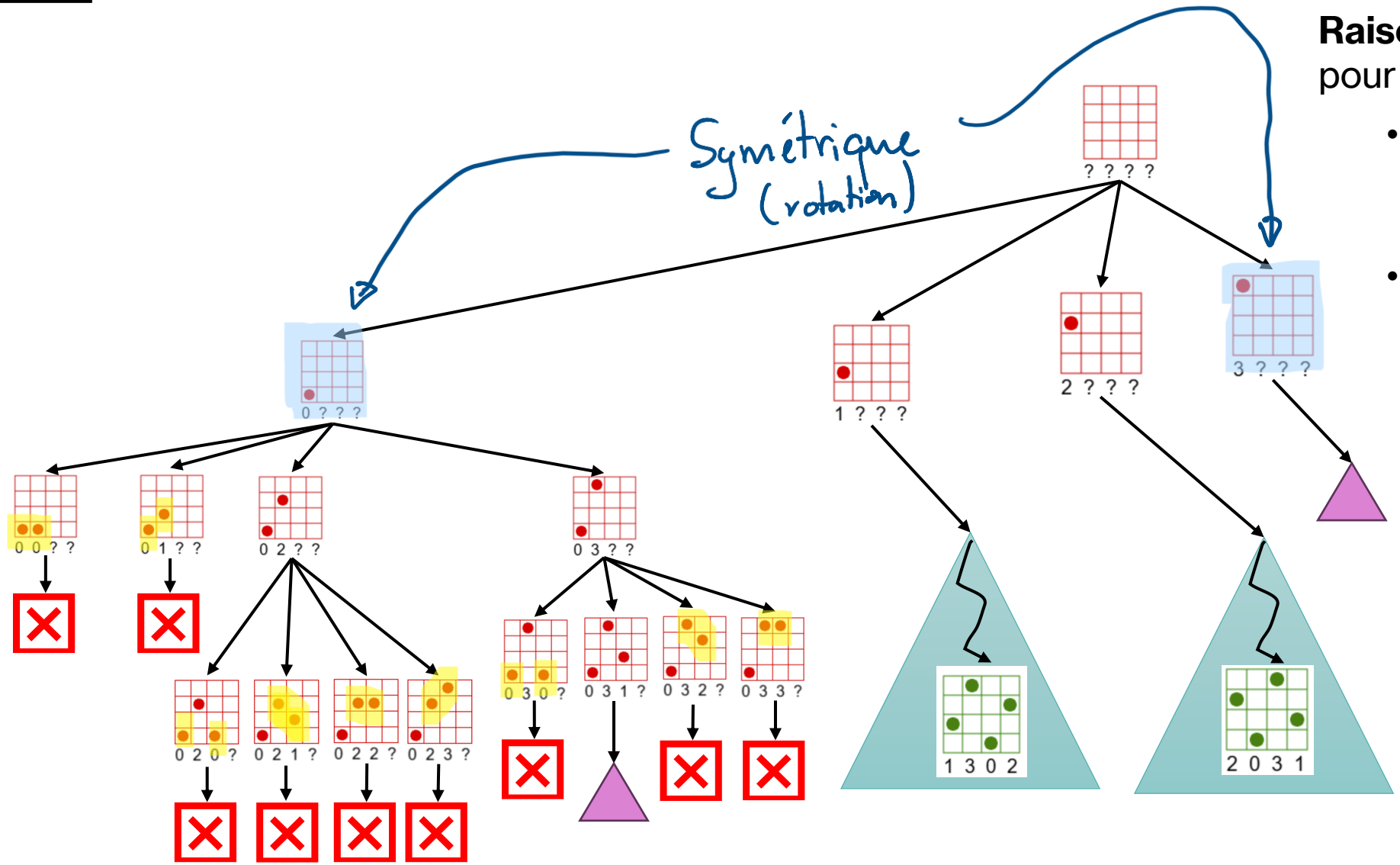
Stratégie 2 : recherche en profondeur + élagage



Amélioration : DFS + élagage



Amélioration : DFS + élagage

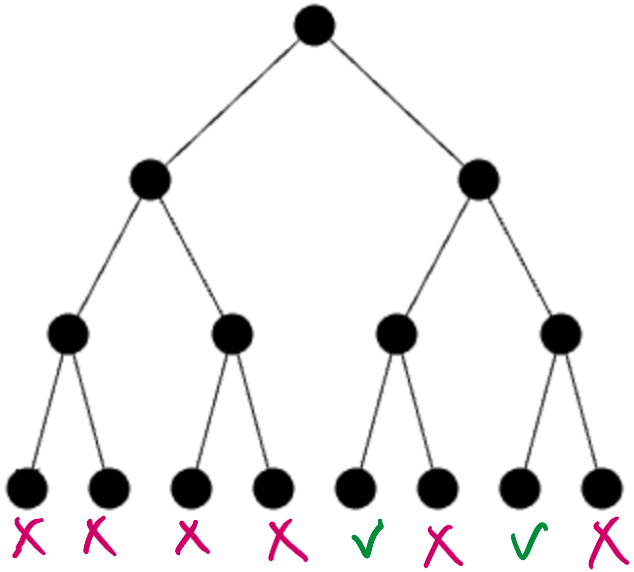


Raisonnements possibles
pour éviter du travail inutile

- Ne pas explorer plus bas dans l'arbre si le parent contient déjà un conflit
- Tenir compte des symétries ➔ si pas de solution pour [0, ?, ?, ?], pas de solution dans le sous-arbre [3, ?, ?, ?]

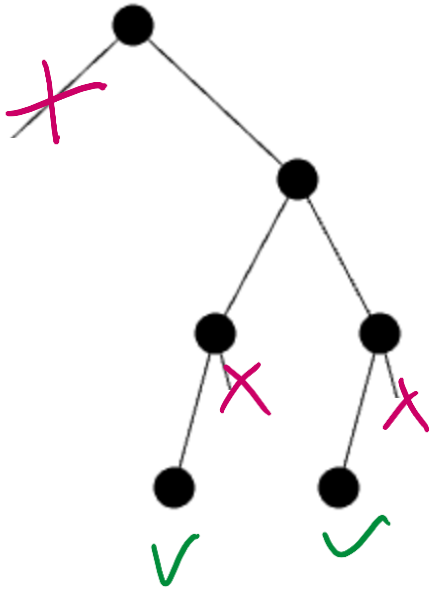
Comparaison entre les deux stratégies

Filtrage
(au niveau des feuilles)



Beaucoup de nœuds à explorer ➔ $O(2^n)$

Élagage
(au niveau des branches)



Beaucoup moins de nœuds à explorer ➔ $O(?)$