# Jeux

**Plan :**

- **Motivations**
- **Formalisation**
- **Classfication de jeux**

Satisfaction de contraintes

# Le taquin

On commence avec une grille 3x3 de neuf cases o`u sont placées huit tuiles étiquetées par les nombres 1...8, une des cases restant vide. Une tuile située `a coté de la case vide peut etre déplacée vers cette case. L’objective du jeu est d’arriver `a obtenir une certaine configuration des tuiles dans la grille.

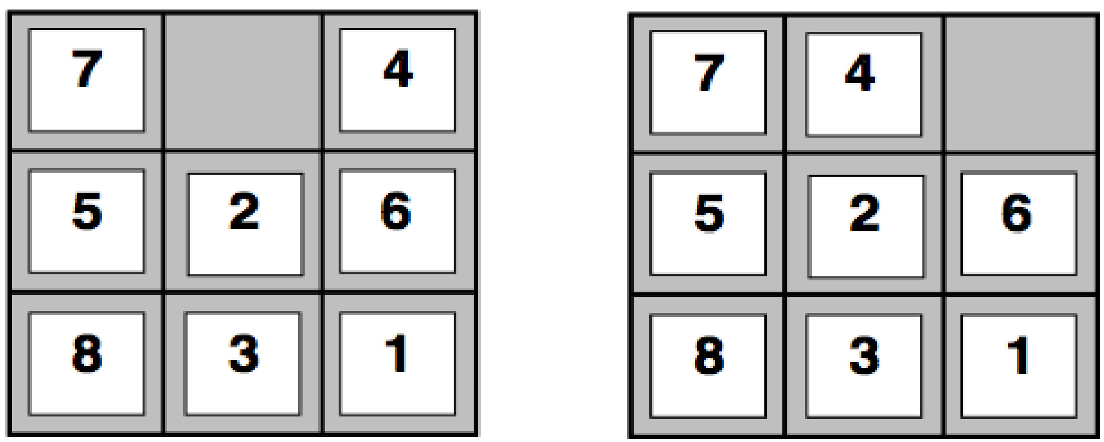



FIG. 1 – Deux états du jeu de taquin.

# Huit reines

Soit le problèmes des n-reines: il faut placer n reines sur un échiquier nXn afin que deux  reines quelconques ne soient pas en prise.

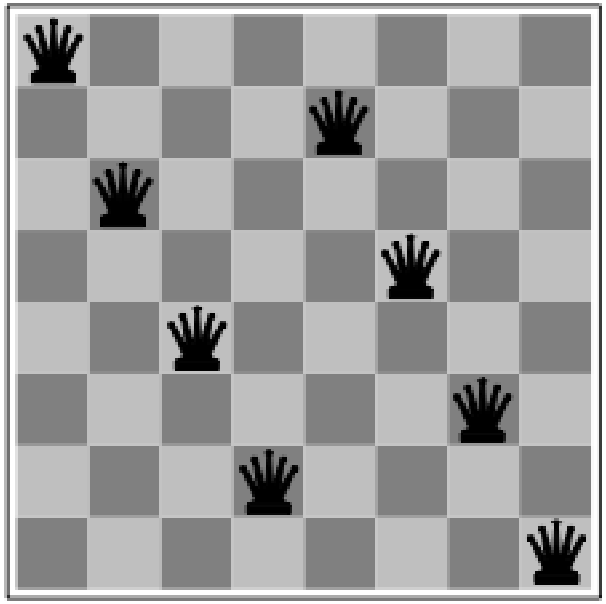


FIG. 2 – Une presque solution pour le problèmes des huit reines.

# La difference entre le taquin et n reines

- Le taquin : nous savon depuis le début quel état nous voulons, et la difficulté est de trouver une séquence d'actions pour l'atteindre

- Huit reines : nous ne sommes pas intéressées par le chemin, mais seulement par l'etat but obtenu.

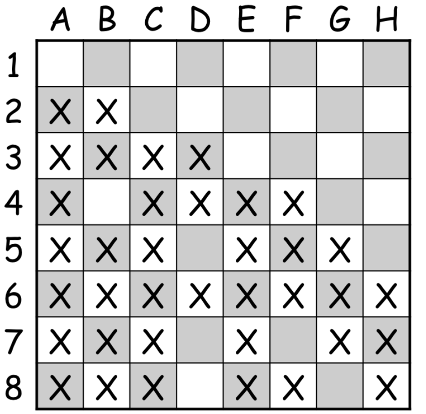
- **Deux grandes classes de juex** :
  - Des problèmes de planification
  - *Des problèmes de satisfaction de contraintes*

# Autre classification de jeux

- Le taquin et n reines ont *un seul agent* qui pouvait choisir librement ses actions
- Jeux a plusieurs joueurs
  - plus compliques parce que l'agent ne peut pas choisir les actions des autres joueurs (cours en 2015)

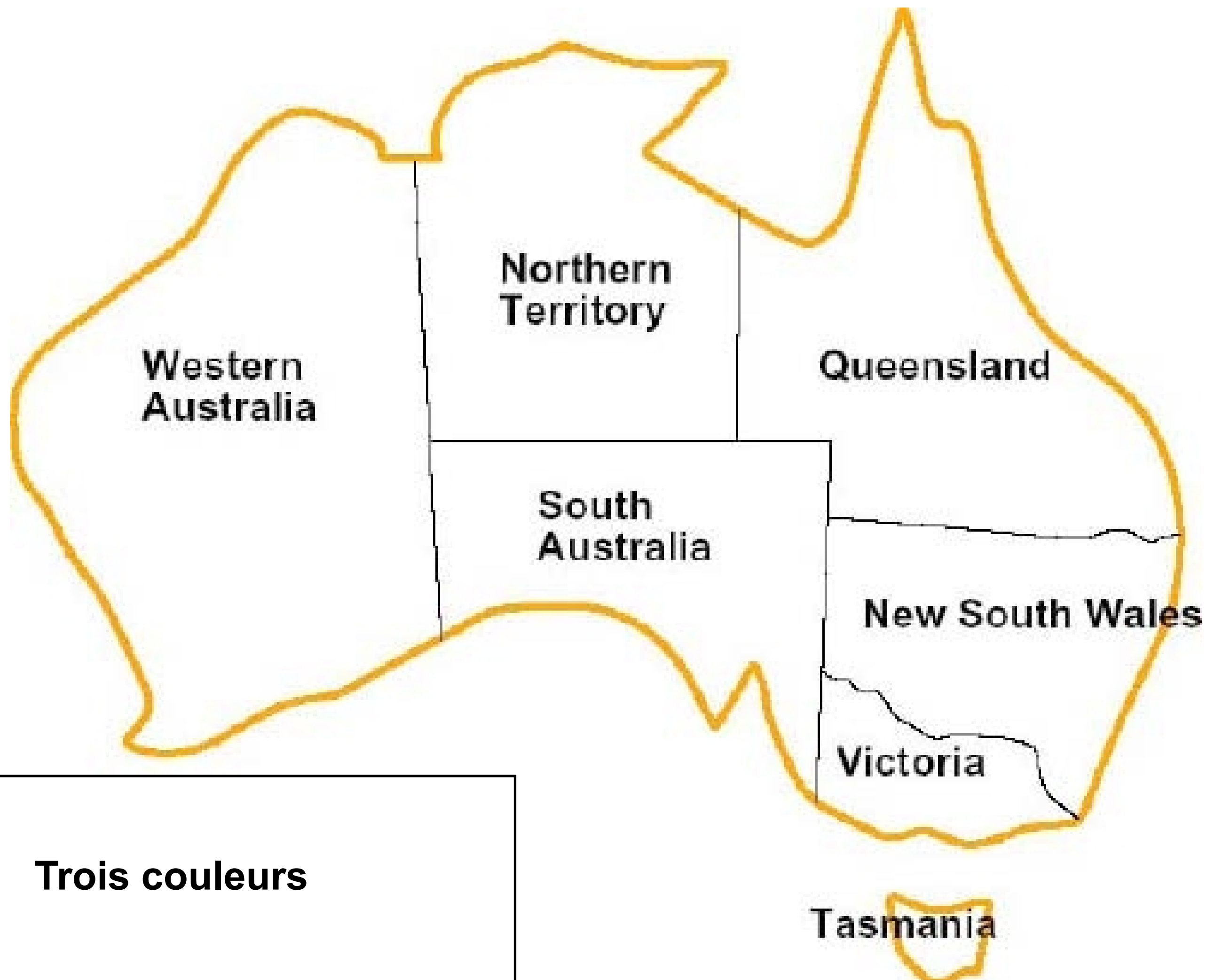
# CSP

# Satisfaction de contraintes



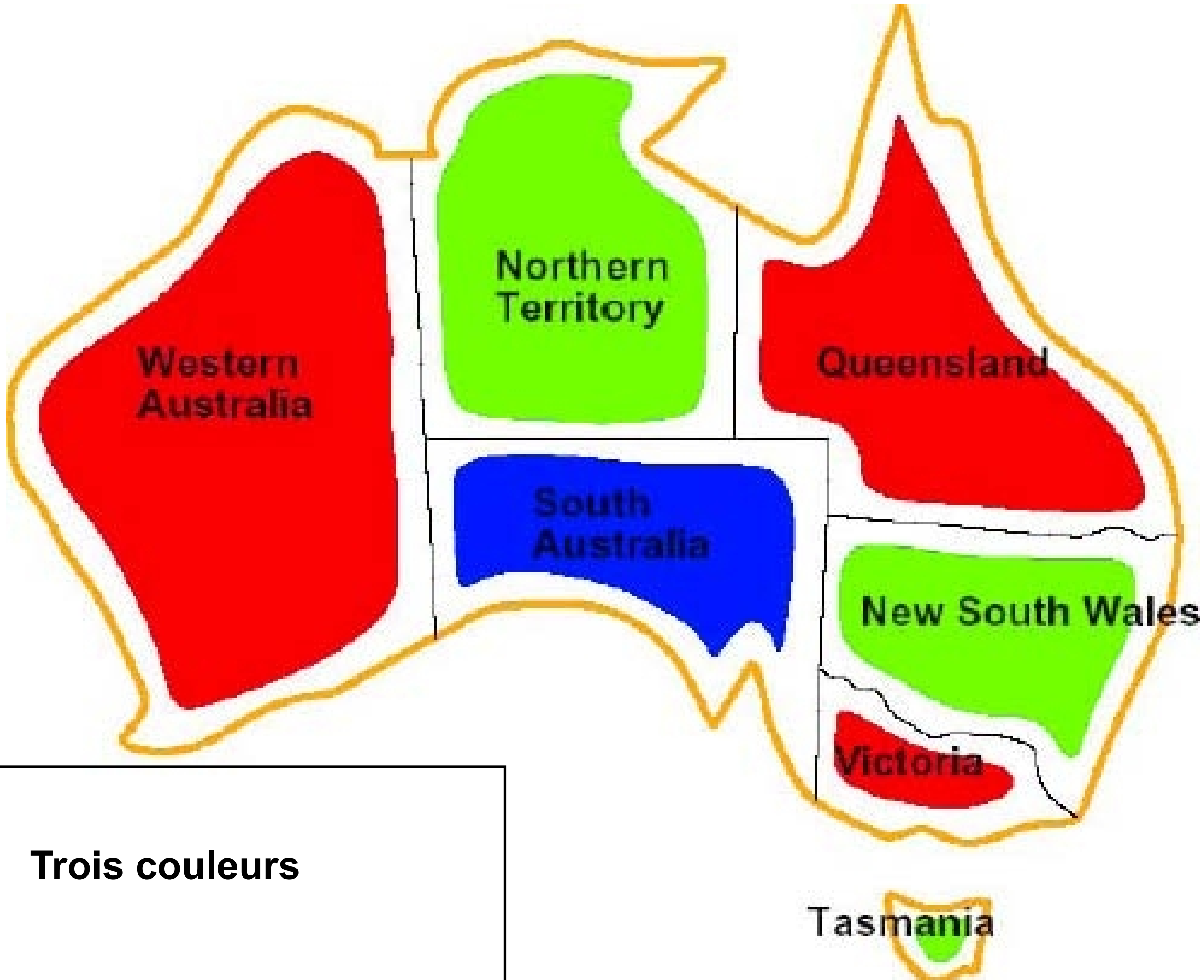

Yue Ma, LRI, Univ. Paris 11

# Les coloriages…



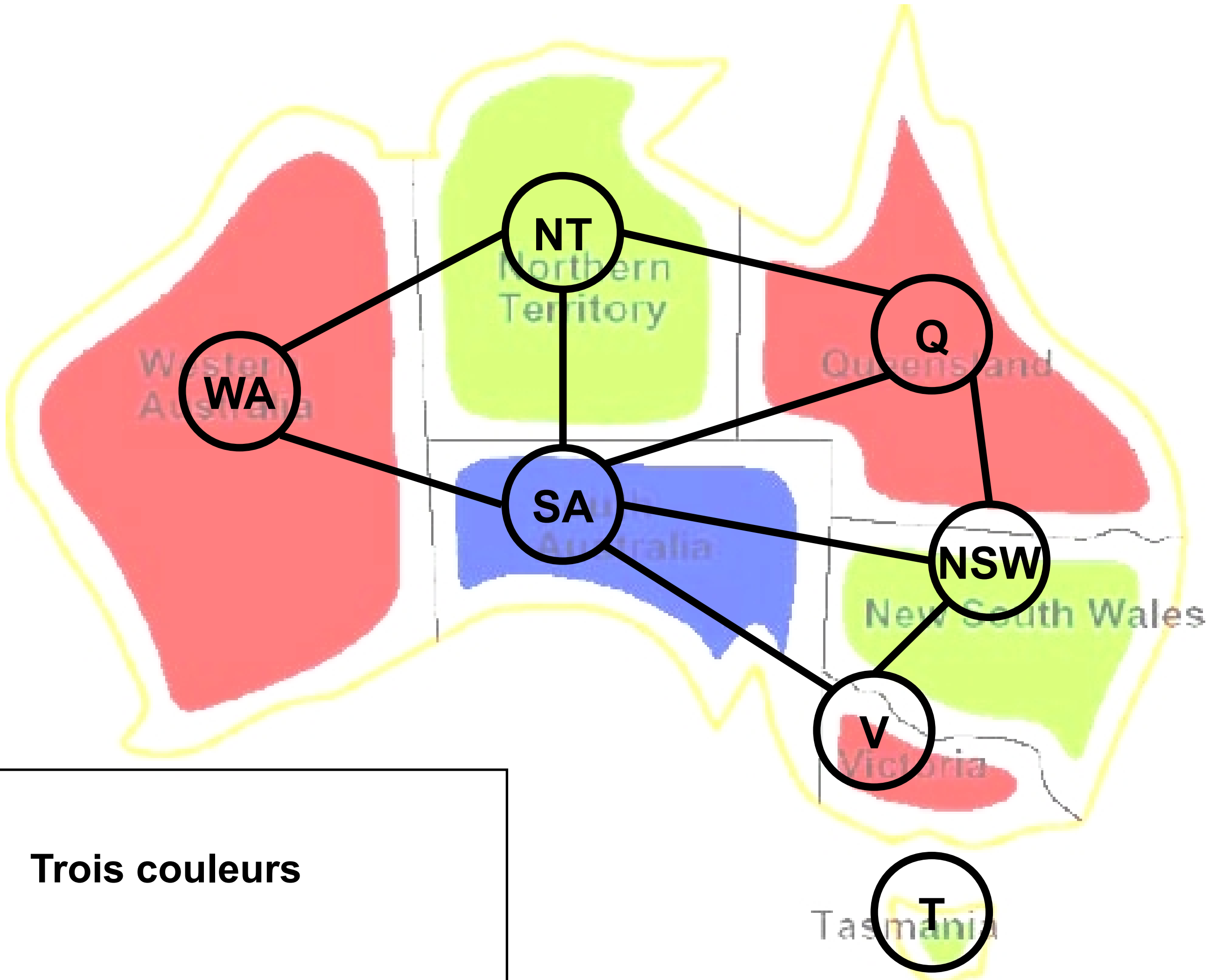

- Trois couleurs
- 2 régions adjacentes ne doivent pas avoir la même couleur

# Les coloriages…



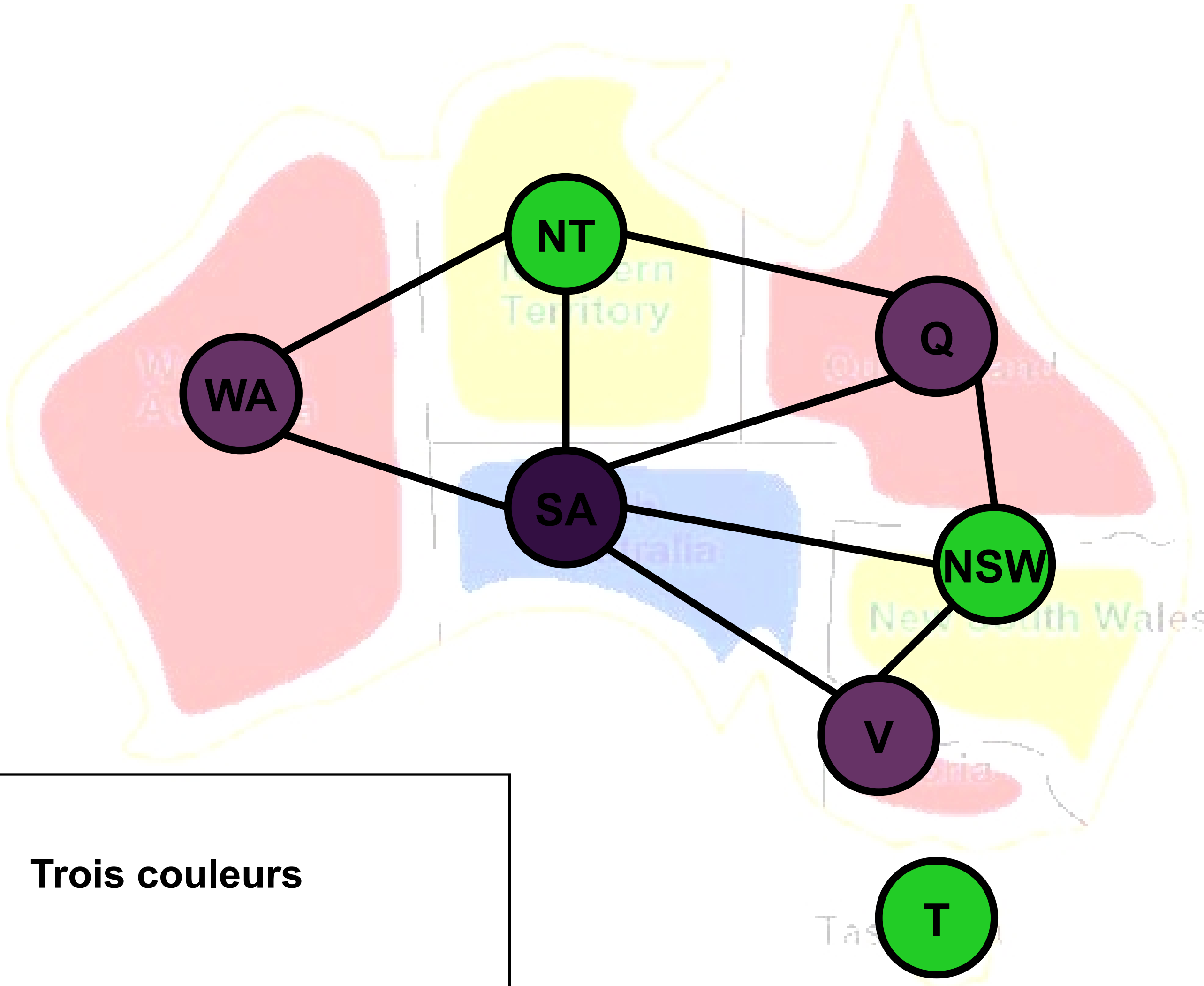

- Trois couleurs
- 2 régions adjacentes ne doivent pas avoir la même couleur

# Les coloriages…




- Trois couleurs
- 2 régions adjacentes ne doivent pas avoir la même couleur

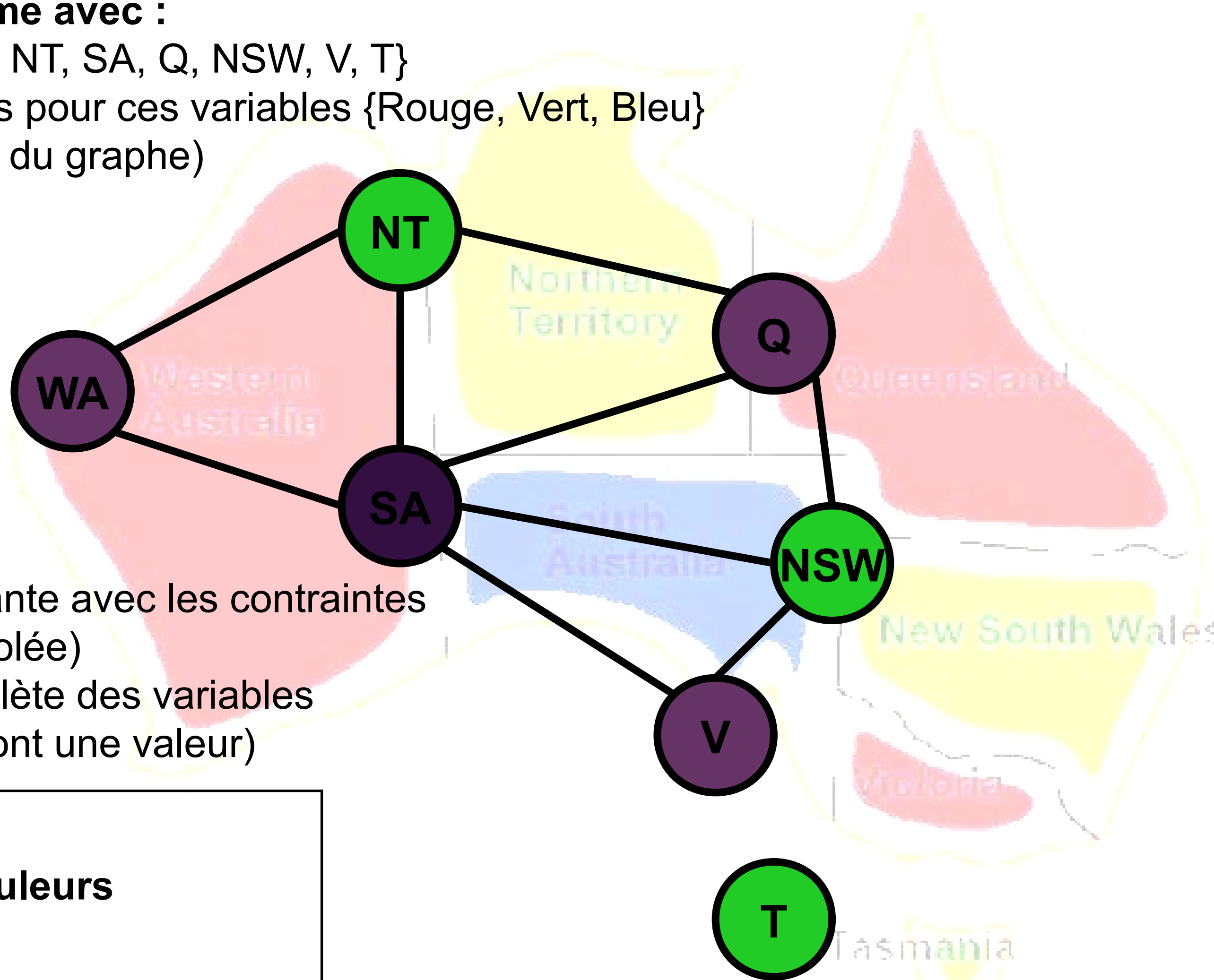
# Les coloriages...




- Trois couleurs
- 2 régions adjacentes ne doivent pas avoir la même couleur

# Les coloriages…

**On a donc un problème avec :**
- Des Variables = {WA, NT, SA, Q, NSW, V, T}
- Des Valeurs possibles pour ces variables {Rouge, Vert, Bleu}
- Des contraintes (arcs du graphe)




**On cherche**
- Une solution consistante avec les contraintes (aucune contrainte violée)
- Une affectation complète des variables (toutes les variables ont une valeur)

- **Trois couleurs**
- **2 régions adjacentes ne doivent pas avoir la même couleur**

# PSC (ou CSP) : Motivations

**Un problème de satisfaction de contraintes :**

- $V = \{V_1, \ldots, V_n\}$ : un ensemble de variables
- $D = \{D_1, \ldots, D_n\}$ : à chaque var X on associe un domaine de val possibles $D_X$
- $C = \{C_1, \ldots, C_p\}$ : un ensemble de contraintes sur un sous-ensemble de V qui restreignent les valeurs que ces variables peuvent prendre **simultanément**.

**Enoncé du problème de satisfaction de contraintes :**

trouver une façon d'instancier chaque variable par une valeur de son domaine qui permette de satisfaire toutes les contraintes de C

**Résolution de problèmes combinatoires :**

- appariement de graphes, coloriage de cartes : problèmes de R.O.
- puzzles (n-reines)
- conception (placement de pièces sur une surface)
- planification, ordonnancement : (emploi du temps, planning PERT...)

# Des variables variables

- Variables discrètes
  - Domaines finis
    - N variables, domaines de taille D
      $O(D^N)$ affectations complètes
    - Ex: PSC booléens
  - Domaines infinis
    - Ex: Ordonnancement de tâches
    - Nécessité d'un langage de contraintes
    - Contraintes linéaires : Soluble
    - Contraintes non linéaires : non décidable
- Variables continues
  - Contraintes linéaires : soluble en temps polynomial par rapport au nombre de variables

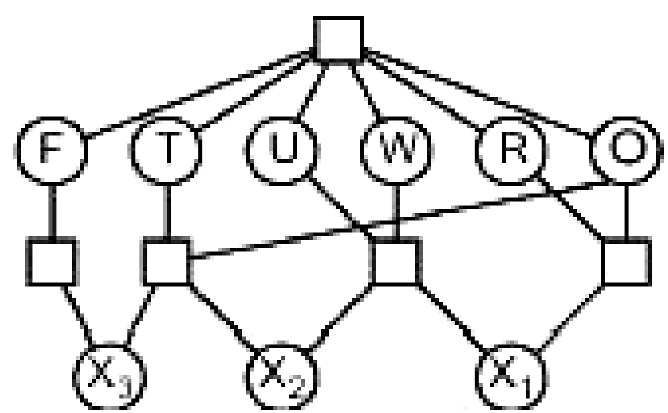
# Types de contraintes

- Unaires
  - Exemple : SA = vert
- Binaires
  - Deux variables (au plus dans chaque contraintes). Ex : SA ≠ WA
- N-aires
  - Les contraintes portent sur 3 variables ou plus (ex : cryptarithmes, mots croisés, …)
- Préférences
  - Plus réaliste. « bleu est mieux que vert »
  - On ne cherche plus 1 solution mais la meilleur

# Exemple de contraintes n-aires :
# Les Cryptarithmes

- Variables { F, T, U, W, R, O}
- Domaines { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Contraintes:
  - AllDiff(F, T, U, W, R, O)
  - $O + O = R + 10 * X_1$
  - $X_1 + W + W = U + 10 * X_2$
  - $X_2 + T + T = O + 10 * X_3$
  - $X_3 = F$

```
    T W O
+   T W O
---------
F   O U R
```



# Dans ce cours

- Des contraintes
  - Linéaires
  - Et Binaires
- Des variables
  - Discrètes
  - Et à Domaines finis

Cela pose déjà quelques problèmes…

# Définitions

(E. Tsang, Principles of Constraint Satisfaction, Academic Press, 1993)

Variable contrainte, domaine

Soit X une variable. On associe à X un ensemble de valeurs possibles pour X, appelé domaine de X, noté Dx

Etiquette

On appelle étiquette une paire <X,v> qui désigne l'assignation à la variable X de la valeur v. L'étiquette <X,v> est valide si v $\in$ Dx

Une k-étiquette composée est l'assignation simultanée de k valeurs à k variables contraintes.

Contrainte

Une contrainte portant sur un ensemble de variables S, noté CS est un ensemble d'étiquettes composées portant sur les variables de S.

Problème de Satisfaction de Contraintes (PSC)

Un triplet (V,D,C) tel que V est un ensemble de variables contraintes, D est l'ensemble des domaines associés à chacune des variables contraintes, C est un ensemble de contraintes portant sur un sous-ensemble arbitraire de V.

# Graphe associé à un PSC

Un ***graphe*** G est un n-uplet (V,U) où V est l'ensemble des noeuds de G et U est l'ensemble des arêtes de G.

Un ***graphe orienté*** G est un n-uplet (V,U) où V est l'ensemble des noeuds de G et U est l'ensemble des arcs de G.

Un ***arc*** ou une ***arête*** du graphe G = (V,U) est une paire de noeuds (X,Y) tel que X et Y $\in$ V. Les noeuds (X,Y) sont ordonnés pour un arc, et ne le sont pas pour une arête. Une arête correspond à deux arcs inverses l'un de l'autre.

Le ***graphe*** (V,U) *associé à un PSC* (V,D,C) est un graphe tel que :
- chaque noeud du graphe est une variable contrainte
- deux noeuds du graphe sont reliés par une arête si deux variables de V apparaissent dans une même contrainte.

**à une contrainte binaire entre deux variables correspondent deux arcs**

# Représentation d'un PSC

**Exemple:**

$V = \{X1, X2, X3, X4, X5\}$

$D_{X1} = \{a, b, c\} = D_{X2} = D_{X5}$

$D_{X3} = \{a, b\} = D_{X4}$

On définit la relation $<$ sur les éléments des domaines; ordre lexicographique strict

Contraintes définies en **intension** :

C1 : $X3 < X1$

C2 : $X3 < X2$

C3 : $X3 < X4$

C4 : $X3 < X5$

C5 : $X4 < X5$

Contraintes définies en **extension** :

C1 : $(X3, X1) \in \{ (a, b), (a, c), (b, c) \}$

C2 : $(X3, X2) \in \{ (a, b), (a, c), (b, c) \}$

C3 : $(X3, X4) \in \{ (a, b) \}$

C4 : $(X3, X5) \in \{ (a, b), (a, c), (b, c) \}$

C5 : $(X4, X5) \in \{ (a, b), (a, c), (b, c) \}$

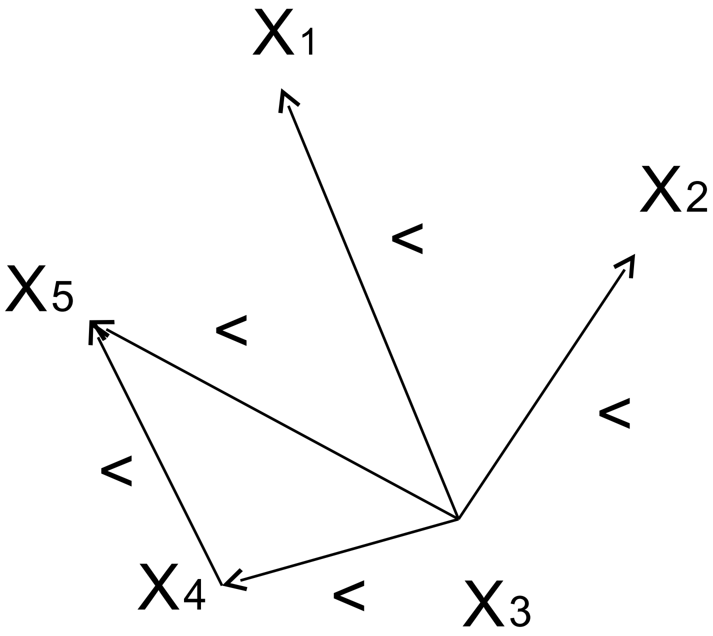
Représentation d'une contrainte par un **tableau** :

| X3 \X1 | a | b | c |
|---|---|---|---|
| a | 0 | 1 | 1 |
| b | 0 | 0 | 1 |

# Représentation d'un PSC

Graphe associé à l'exemple :

$D_{X1} = D_{X2} = D_{X5} = \{ a, b, c \}$
$D_{X3} = D_{X4} = \{a, b \}$




C1 : X3 < X1
C2 : X3 < X2
C3 : X3 < X4
C4 : X3 < X5
C5 : X4 < X5

# Problématique

On peut chercher :

- toutes les solutions
- une solution
- la solution optimale ou une solution proche de l'optimale

**Techniques de réduction de problème**

Problème de satisfaction de contrainte P (V,D,C)

- bien formulé (de l'art de formuler un problème de contraintes...):
  - contraintes redondantes, partielles
  - domaines mal ajustés (trop larges ou trop restreints)
  - variables mal choisies....

**Recherche excessivement coûteuse**

- reformulation partielle du problème en un problème équivalent (même ensemble de solutions) mais plus simple à résoudre

# Problématique (suite)

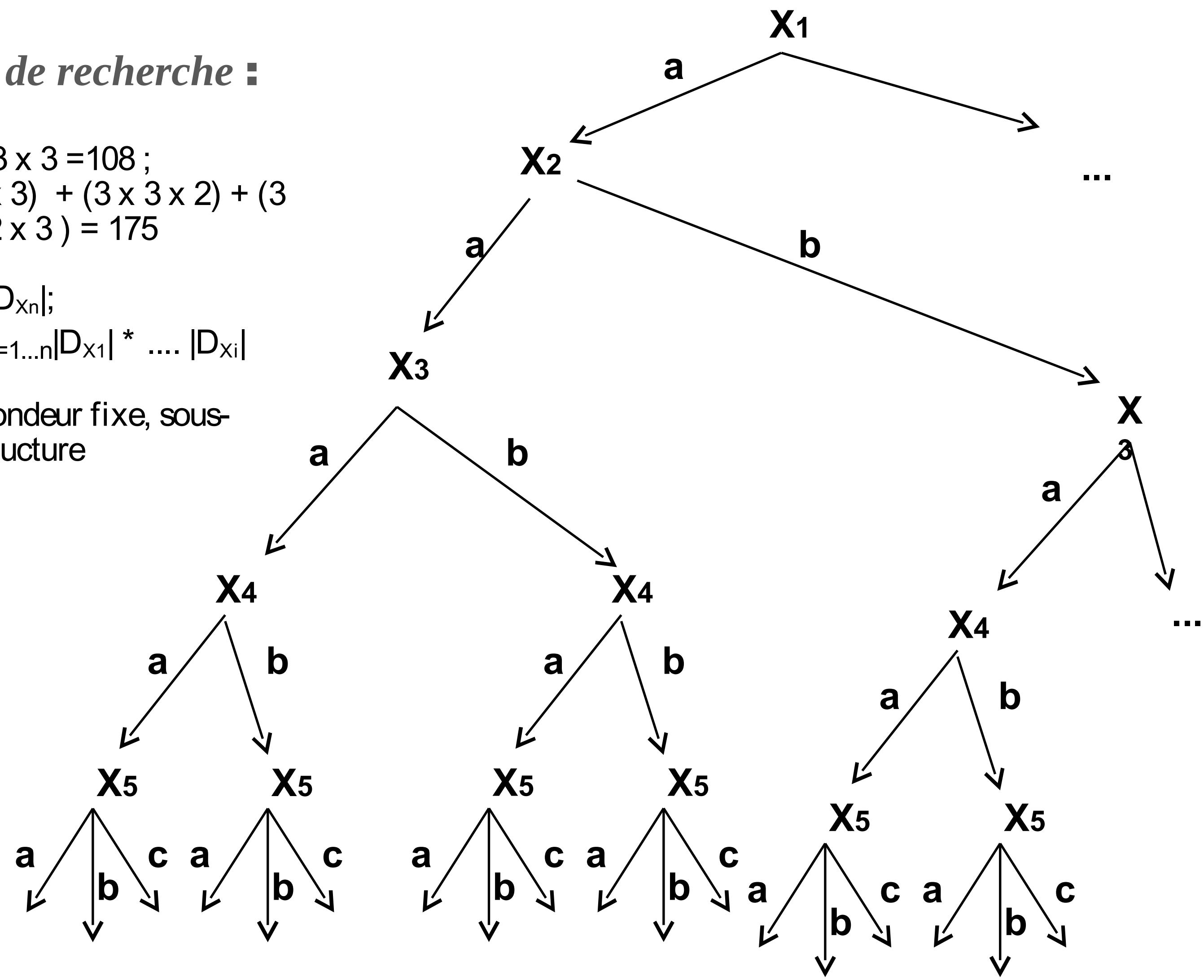
**Techniques de recherche à la résolution de PSC**

***Structure d'un arbre de recherche*** **:**

Nb de feuilles : 2 x 2x 3 x 3 x 3 =108 ;
Nb de noeuds : 1 + 3 + (3 x 3) + (3 x 3 x 2) + (3 x 3 x 2 x 2) + (3 x 3 x 2 x 2 x 3 ) = 175

Nb de feuilles : $|D_{X1}|$ * .... $|D_{Xn}|$;
nb de noeuds : 1 + $\mathrm{Sum}_{i=1...n}|D_{X1}|$ * .... $|D_{Xi}|$

Taille de l'arbre finie, profondeur fixe, sous-arbres "frères" de même structure



# Problèmes abordés

**Points de choix dans la recherche** : dans quel ordre

- affecter les variables
- affecter les valeurs à une variable
- tester les contraintes

- Techniques permettant de mieux **répartir l'effort de recherche** sur les points de choix

- Techniques de **retour arrière « intelligent »**

- Techniques **d'apprentissage** en cours de recherche : optimiser l'exploration des sous arbres frères par l'analyse des échecs en cours de recherche.

# Définitions (suite)

**Projection**

Une m-étiquette composée M est la *projection* d'une n-étiquette composée N ($m \leq n$) ssi toutes les étiquettes de M apparaissent dans N

***exemple*** : Soit la 3-étiquette composée N = { <X, 1>, <Y, 2>, <Z, 3> } .
La 2 -étiquette composée M = { <Y, 2>, <Z, 3> } est une projection de N sur Y et Z**.**

**Satisfiabilité d'une contrainte**

Une étiquette composée E *satisfait* une contrainte C ssi la projection de E sur les variables de C appartient à C. Dans le cas contraire, E *viole* C.

***exemple*** (ordre lexicographique suite): Soit la 3-étiquette
E = { <X1, b >, <X2, b >, <X3, a > } et C1 : X3 < X1.
La projection de E sur les variables X1 et X3 est
E' = { <X1, b >, <X3, a > } . E' appartient à C1, donc E satisfait C1.
**Mais** la 3-étiquette F = { <X1, b >, <X2, b >, <X3, b > } viole C1.

# Définitions (suite)

S'il existe au moins une étiquette composée qui satisfasse C, C est dite satisfiable. Sinon C est dite insatisfiable.

**Solution d'un PSC (V,D,C) :**

Si |V|=n, une solution est une n-étiquette composée qui satisfait toutes les contraintes de C.

**Exemple** (ordre lexicographique suite):

La 5-étiquette composée

E = { <X1, b >, <X2, c >, <X3, a >, <X4, b >, <X5, c > }

est une solution du PSC.

# Algorithme standard pour la satisfaction de contraintes

## Algorithme de retour arrière

**Notations :**

- **NE : Ensemble des variables non étiquetées**
- **EC Etiquette composée courante (ensemble des variables déjà affectées)**
- **D: Ensemble des domaines des variables contraintes du PSC à résoudre**
- **C : Ensemble des contraintes du PSC à résoudre**

**Procédure retour_arrière(V, D, C)**
**Res $\leftarrow$ backtrack(V, $\varnothing$ , D, C);**
**return(Res)**

**Procédure backtrack(NE, EC, D, C)**
**Si NE = $\varnothing$ alors return(EC)**
**sinon**
    **Sélectionner une variable X dans NE;**
    **Répéter**
        **Sélectionner une valeur v dans $D_x$; $D_x \leftarrow D_x - \{ v \}$;**
        **si EC $\cup$ <X, v> ne viole pas les contraintes de C**
            **alors**
            **Résultat$\leftarrow$ backtrack( NE - { X }, EC $\cup$ <X, v>, D, C)**
            **si Résultat $\neq \varnothing$ alors return(Résultat)**
        **finsi**
    **jusqu'à ce que $D_x = \varnothing$**
**return( $\varnothing$ )**
**finsi**

# Critique de l'algorithme standard

L'algorithme souffre de nombreux défauts :

- parcours aveugle de l'espace sans prise en compte d'informations évidentes sur la non satisfiabilité globale d'instanciations partielles,
- redécouverte répétée des mêmes insatisfiabilités locales...

==>> **algorithmes plus efficaces** : trois approches

- définitions de problèmes polynomiaux pour des propriétés de cohérence affaiblie : arc-cohérence ...
- améliorations de l'algorithme standard de backtrack
- définition de méthodes de décomposition de PSC en sous-PSC plus simples, si possible, dont la satisfiabilité soit un problème polynomial

# Réduction de problèmes

- But :

Soit un PSC P = (V,D,C). Transformer P en P' = (V,D',C') tel que P et P' soient équivalents et que P' soit plus « simple » à résoudre que P.

- Deux PSC sont *équivalents* s'ils portent sur le même ensemble de variables et qu'ils ont le même ensemble de solutions.

- Soient deux PSC P = (V,D,C) et P' = (V,D',C'). P' est une *réduction* de P ssi
  - P et P' sont équivalents
  - pour toute variable X de V, dom(X)¿ ¿ ¿ dom'(X).
  - C subsume C'.

# Réduction de problèmes (suite)

Donc intuitivement, on réduit P en P' en enlevant

- des domaines des variables dans D des valeurs qui n'apparaissent dans aucun tuple solution
- des contraintes de C des k-étiquettes composées qui ne sont la projection d'aucun tuple solution

On va étudier dans le cours 3 propriétés

cohérence par noeud < cohérence par arc < cohérence par chemin

Attention = cohérence ne veut pas dire satisfiabilité (un PSC cohérent par noeud, arc ou chemin n'a pas forcément de solution).

# Cohérences locales : introduction

On voit des contraints de facon passif (comme un test) ...

Pouvons-nous utiliser les contraintes d'une manière plus active?

**Exemple:**

$$\mathbf{A} \in \{3, \ldots, 7\}, \quad \mathbf{B} \in \{1, \ldots, 5\}$$

$$A < B$$

Plusieurs values incohérentes pouvent etre enleve

On obtient : $\mathbf{A} \in \{3, \quad 4\}, \quad \mathbf{B} \in \{4, \quad 5\}$

Attention : il ne garantie pas que les combinations de toute les reste values pour chaque variable est cohérentes. Par exemple A=4, B=4 ne l'est pas.

Comment on peut enlever les values incohérentes **des domaines des variables ?**

# Cohérence par noeud

Un PSC (V,D,C) est ***cohérent par noeud*** si pour toute variable X de V, toutes les valeurs de $D_X$ satisfont les contraintes unaires sur X. (*1-cohérence*)

**Algorithme**

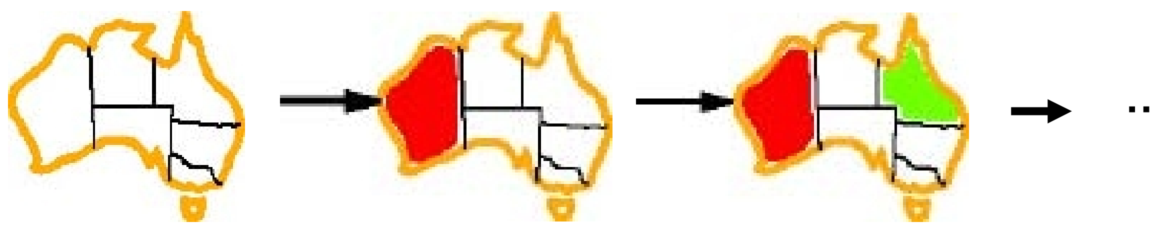
```
PROCEDURE NC(V,D,C)

Pour tout X ∈ V
    Pour tout a ∈ Dx
        si <X, a > ne satisfait pas Cx
            alors Dx ← Dx - { a }
    finpour
finpour
retourner(V,D,C)
```

**Complexité** : n = |V|, a = max(|$D_i$|),
(une seule contrainte unaire par variable, au plus)
linéaire en nb de variables contraintes (O(an)).

# Cohérence des arcs

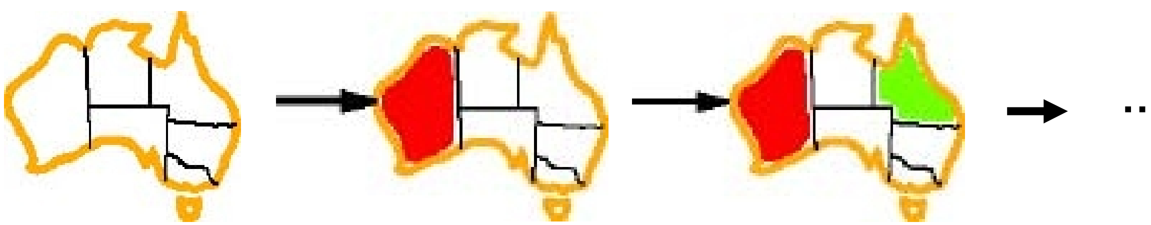
- Un arc entre X et Y est cohérent si pour toutes valeurs x de X, il y a au moins une valeur possible y de Y

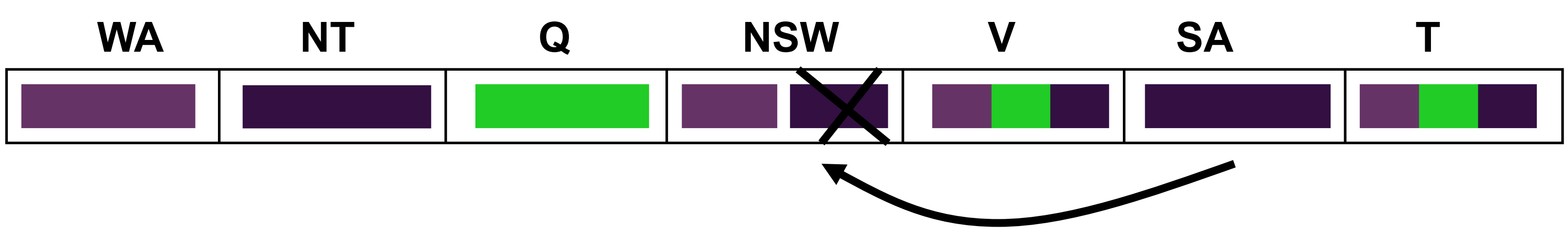

**A un moment de la recherche :**

# Cohérence des arcs


- Un arc entre X et Y est cohérent si pour toutes valeurs x de X, il y a au moins une valeur possible y de Y

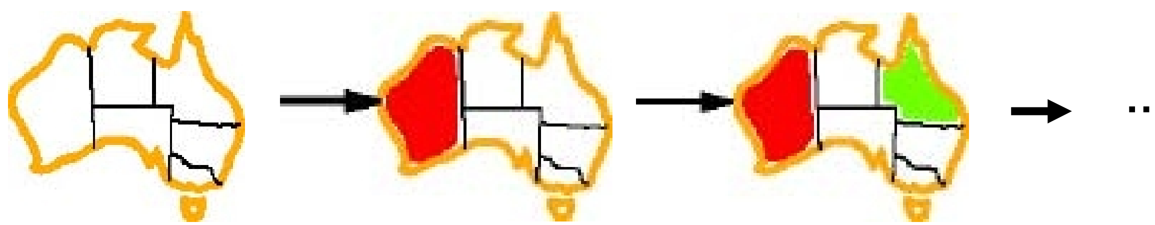



**A un moment de la recherche :**

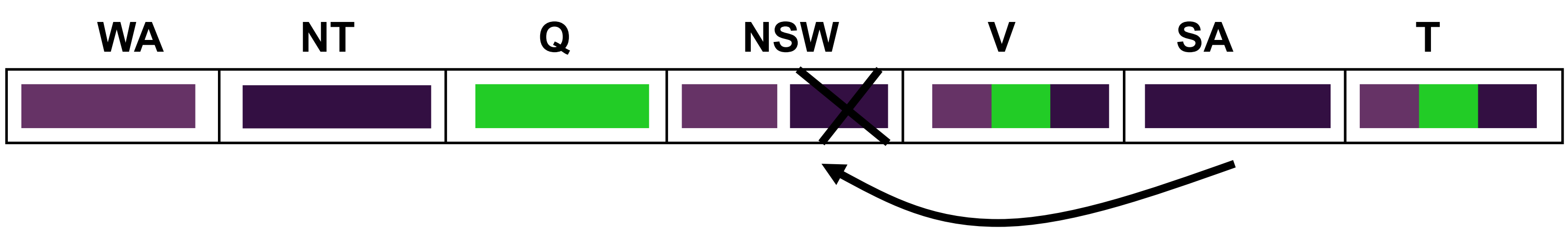


# Cohérence des arcs

- Un arc entre X et Y est cohérent si pour toutes valeurs x de X, il y a au moins une valeur possible y de Y

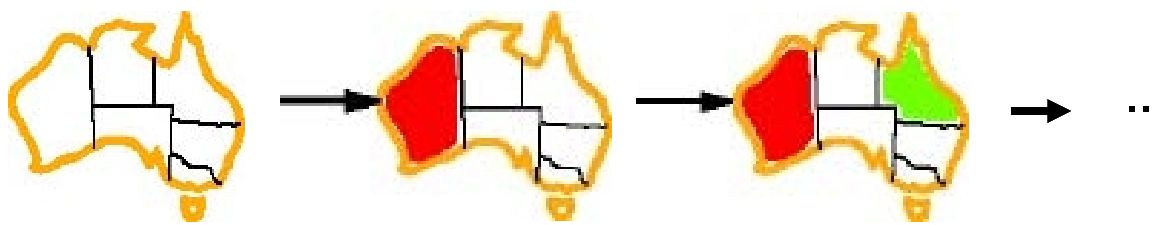




**A un moment de la recherche :**

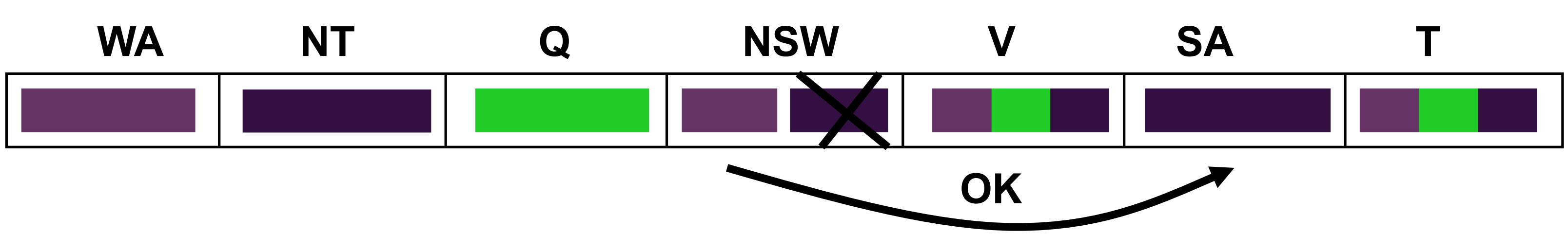


# Cohérence des arcs

- Un arc entre X et Y est cohérent si pour toutes valeurs x de X, il y a au moins une valeur possible y de Y

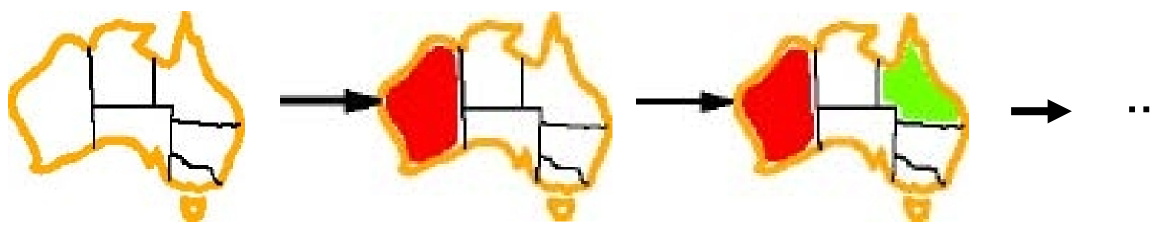




**A un moment de la recherche :**

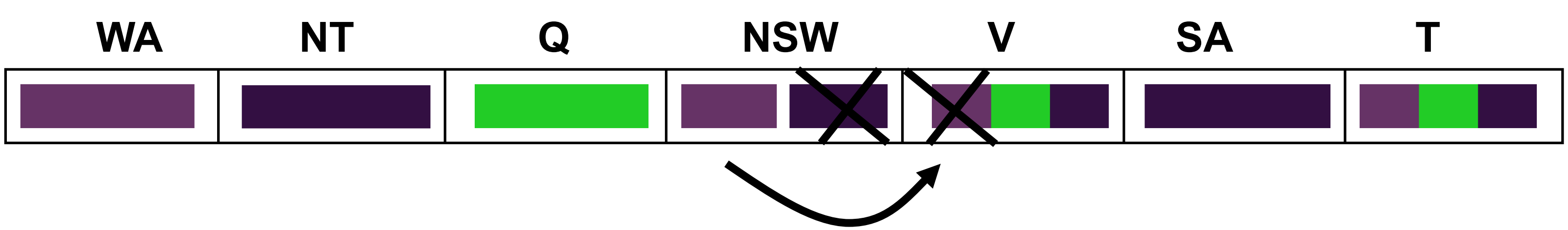


# Cohérence des arcs

- Un arc entre X et Y est cohérent si pour toutes valeurs x de X, il y a au moins une valeur possible y de Y

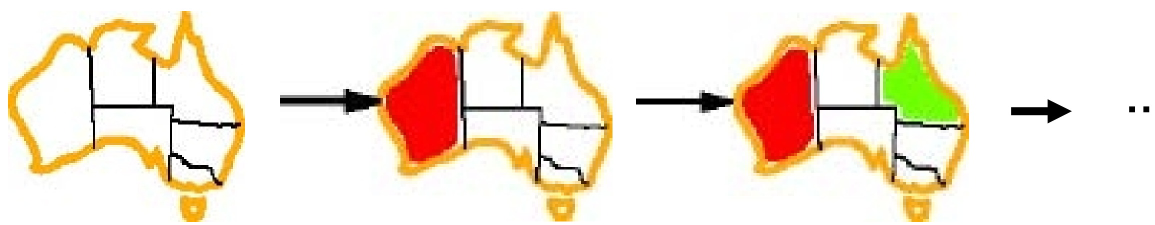




**A un moment de la recherche :**



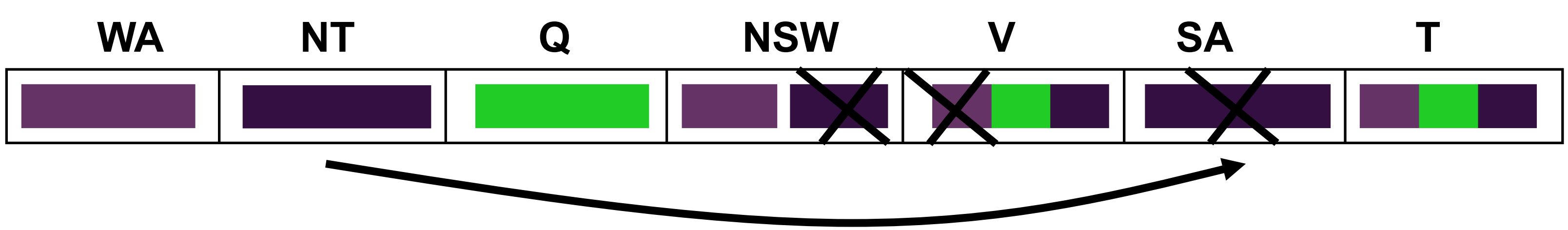
# Cohérence des arcs

- Un arc entre X et Y est cohérent si pour toutes valeurs x de X, il y a au moins une valeur possible y de Y


**A un moment de la recherche :**

**Inconsistance détectée !**



# Cohérence par arc

- Un arc (X,Y) dans un graphe de contraintes est *cohérent* ssi pour toute valeur $a \in D_X$ qui satisfait les contraintes unaires sur X, il existe une valeur $b \in D_Y$ qui est compatible avec <X, a>, c'est-à-dire telle que {<X, a>,<Y, b>} satisfasse les contraintes binaires portant sur X et Y (*2-cohérence*).
- Un PSC (V,D,C) est *cohérent par arc* ssi tout arc dans le graphe de contraintes de (V,D,C) est cohérent.
- Peut être exécuté comme pré-processus ou après chaque assignation

Attention à l'asymétrie de X et Y

**PROCEDURE réduit_domaine((X,Y),(V,D,C))**
**Efface ← faux;**
**Pour chaque $a \in D_X$**
**s'il n'existe pas de $b \in D_Y$ t.q. {<X, a >,<Y, b >} satisfasse $C_{XY}$**
**alors $D_X \leftarrow D_X - \{a\}$; Efface ← vrai;**
**finsi**
**finpour**
**return(Efface) /* Efface indique si le domaine de X a été modifié */**

# Comment rendre CSP *cohérent par arc* ?

- Faire **réduit_domaine((X,Y),(V,D,C)) pour chaque arc (X,Y) ?**
- **Mais, cela n'est pas suffit !** Pruning the domain already revised May Make Some arcs inconsistent again.

A<B, B<C: (3…7,1…5,1…5)

(**3…4**,1…5,1…5)

(3…4,4…5,1…5)

(3…4,4,1…5)

(3…4,4,5)

(3,4,5)

Donc, **réduit_domaine doit continuer jusqu'`a le domain ne change plus.**

# Cohérence par arc (suite)

Plusieurs manières d'implanter l'algorithme

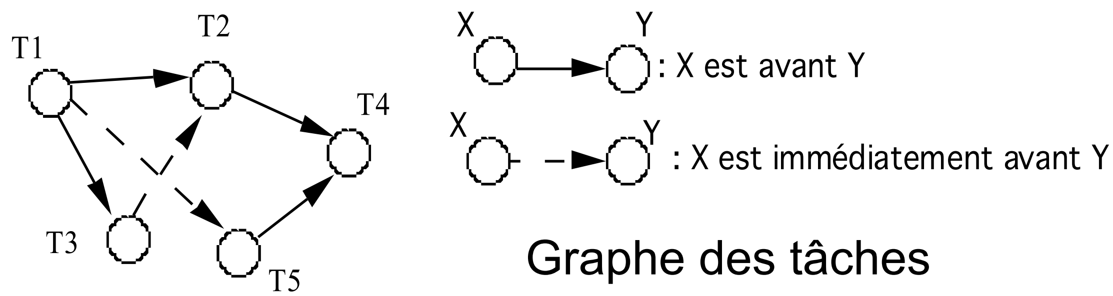
La manière brutale : (Mackworth 1977)

```
PROCEDURE AC-1(V,D,C)
NC(V,D,C); /* on s'assure d'abord de la cohérence par noeud */

File ←  {(X,Y) et (Y,X) | C_XY est une contrainte binaire i.e. X≠  Y}
Repeter
    modif ←  faux;
    pour chaque arc (X,Y) de File
        modif ← (reduit_domaine((X,Y),(V,D,C)) ou modif )
    finpour
jusqu'à ce que modif = faux
retourner(V,D,C)
```

# Exemple de déroulement de AC-1

**DTi = [1..4]**
**C1 : T1 < T2**
**C2 : T1 < T3**
**C3 : T2 < T4**
**C4 : T5 < T4**
**C5 : T1 + 1 = T5**
**C6 : T3 + 1 = T2**
**C7 : T2 < 4**




Graphe des tâches

**<u>Après NC :</u> DT2 ← [1..3] (une seule contrainte unaire : C7)**
**<u>Initialisation de File :</u> File := {(T1,T2), (T1,T3), (T2,T4), (T5,T4), (T1,T5), (T3,T2), (T2,T1), (T3,T1), (T4,T2), (T4,T5), (T5,T1), (T2,T3) }**

**<u>Première itération sur File :</u>**

- **Traitement (T1,T2) : DT1 ← [1.. 2]; modif ← vrai (pour toute l'itération sur File)**
- **Traitement (T1,T3) : DT1 inchangé**
- **Traitement (T2,T4) : DT2 inchangé**
- **Traitement (T5,T4) : DT5 ← [1.. 3]**
- **Traitement (T1,T5) : DT1 inchangé (contrainte C5)**
- **Traitement (T3,T2) : DT3 ← [1.. 2] (contrainte C6)**
- **Traitement (T2,T1) : DT2 ← [2.. 3] (on commence la 2ème série d'arcs dans File)**

# Exemple de déroulement de AC-1

- **Traitement (T3,T1) : <u>DT3 ← 2</u>**
- **Traitement (T4,T2) : DT4 ← [3.. 4]**
- **Traitement (T4,T5) : DT4 inchangé**
- **Traitement (T5,T1) : DT5 ← [2.. 3]**
- **Traitement (T2,T3) : <u>DT2 ← 3</u>**

**<u>Deuxière itération sur File</u>**

- **Traitement (T1,T2) : DT1 inchangé**
- **Traitement (T1,T3) : <u>DT1 ← 1</u>; modif ← vrai (pour toute l'itération sur File)**
- **Traitement (T2,T4) : DT2 inchangé**
- **Traitement (T5,T4) : DT5 inchangé**
- **Traitement (T1,T5) : DT1 inchangé**
- **Traitement (T3,T2) : DT3 inchangé**
- **Traitement (T2,T1) : DT2 inchangé**
- **Traitement (T3,T1) : T3 inchangé**
- **Traitement (T4,T2) : <u>DT4 ← 4</u>**
- **Traitement (T4,T5) : DT4 inchangé**
- **Traitement (T5,T1) : <u>DT5 ← 2</u>**
- **Traitement (T2,T3) : DT2 inchangé**

**<u>Troisième itération sur File :</u> Pas de domaine réduit; AC-1 termine.**

# Commentaires sur AC-1

**AC-1:** étant donné un PSC P = (V,D,C), AC1 rend un PSC P' = (V,D',C) qui est équivalent à P (la réduction des domaines ne supprime aucune solution potentielle) et cohérent par noeud et par arc

Dans le cas général, le problème initial P n'est pas pour autant résolu sauf si :

- il existe au moins un $X \in V$ tq son domaine réduit est $\emptyset$ : P est insatisfiable
- tous les domaines réduits sont des singletons sauf un qui est de cardinal n.

  ==>> dans ce cas, le problème P a n solutions.

**Complexité :**

soit e = nb arêtes dans le graphe de P, n = |V|, a = max(|$D_i$|),

dans le pire des cas: une seule valeur d'une seule variable est éliminée à chaque itération (n itérations); à chaque itération, il y a au pire 2 e appels de reduit_domaine (car |File| = 2e); un appel de reduit-domaine examine $a^2$ valeurs

==>> complexité en $O(a^3ne)$

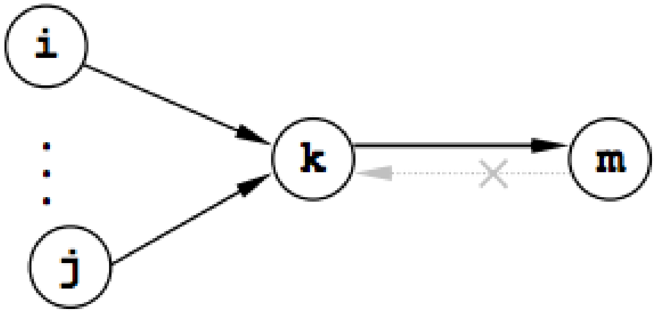
**Inacceptable, surtout si les domaines des variables contraintes sont grands**

# What's wrong with AC-1

If a single domain is pruned then revisions of al the arcs are repeated even if the pruned domain does not influence most of these arcs.

What arcs should be reconsidered for revisions?

The arcs whose consistency is affected by the domain pruning i.e. the arcs pointing to the changed variable: $(V_i, V_k) \ldots (V_j, V_k)$ need to be revised.




If we revise the arc $(V_k, V_m)$ and the domain of $V_k$ is reduced, it is NOT necessry to re-revise the arc $(V_m, V_k)$ !!!

--- because deleted values of $V_k$ are NOT support for any valued in the current domain of Vk (are not consistent with values of Vm, so they are deleted when visiting (Vm, Vk))

# AC-2 (Mackworth 1977)

- In every step, the arcs going back from a given vertex are processed
- Data structure is not optimal → AC-3

# Algorithme AC-3 (Mackworth 1977)

Dès qu'il y a réduction du domaine d'une contrainte (modif est Vrai), toutes les arêtes du graphe sont à nouveau explorées.

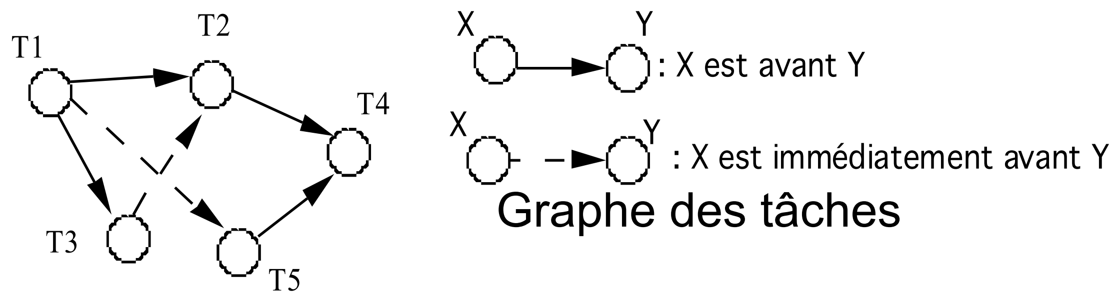
Sur l'exemple précédent, beaucoup d'arêtes explorées qui ne mènent à aucune réduction de domaine. Pourquoi ?

Quand le domaine d'une variable contrainte X est réduit, quelles sont les variables susceptibles d'être affectées ? Celles qui sont connectées à X (il existe une arête $(X,X_i)$ dans le graphe de contraintes associé à P).

```
PROCEDURE AC-3(V, D, C)
NC(V,D,C); /* on s'assure d'abord de la cohérence par noeud */
File := {(X,Y) et (Y,X)  / C_XY est une contrainte binaire i.e. X ≠ Y}
Tant que File ≠ ∅
     File ← File - { (X,Y) }
     si reduit_domaine((X,Y), (V, D, C))
               alors File ← File ∪ {(Z,X) | C_ZX ∈ C, Z≠ X et Z≠ Y}
     finsi
fintantque
return(V, D, C)
```

# Exemple de déroulement de AC-3

**DTi = [1..4]**
**C1 : T1 < T2**
**C2 : T1 < T3**
**C3 : T2 < T4**
**C4 : T5 < T4**
**C5 : T1 + 1 = T5**
**C6 : T3 + 1 = T2**
**C7 : T2 < 4**




Graphe des tâches

**<u>Après NC :</u> DT2 ← [1..3] (une seule contrainte unaire : C7)**

**<u>Initialisation de File :</u>= {(T1,T2), (T1,T3), (T2,T4), (T5,T4), (T1,T5), (T3,T2),**

**(T2,T1), (T3,T1), (T4,T2), (T4,T5), (T5,T1), (T2,T3) }**

- **Traitement (T1,T2) : DT1 : [1..2]; (T3,T1) et (T5,T1) déjà dans File, File ← File - (T1,T2)**
- **Traitement (T1,T3) : DT1 inchangé; File ← File - (T1,T3)**
- **Traitement (T2,T4) : DT2 inchangé ; File ← File - (T2,T4)**
- **Traitement (T5,T4) : DT5 : [1.. 3]; File ← File - (T5,T4)**

# Exemple de déroulement de AC-3

• (T2,T1) : DT2 : [2.. 3]; File ← File - (T2, T1) ∪ (T3,T2)

File ← { (T3,T1), (T4,T2), (T4,T5), (T5,T1), (T2,T3), (T1,T3), (T3,T2)}

• (T3,T1) : <u>DT3 = 2</u>; File ← File - (T3,T1)

• (T4,T2) : DT4 : [3..4]; File ← File - (T4,T2) ∪ (T5,T4)

• (T4,T5) : DT4 inchangé; File ← File - (T4,T5)

• (T5,T1) : DT5 : [2..3]; File ← File - (T5,T1) ∪ (T4,T5)

• (T2,T3) : <u>DT2 = 3</u>; File ← { (T1,T3), (T3,T2), (T5,T4), (T4,T5), (T1, T2), (T4,T2)}

• (T1,T3) : <u>DT1 = 1</u>; File ← {(T3,T2), (T5,T4),(T4,T5), (T1, T2), (T4,T2),(T2, T1), (T5,T1)}

• (T3,T2) : DT3 inchangé; File ← {(T5,T4), (T4,T5), (T1, T2), (T4,T2), (T2, T1), (T5,T1)}

• (T5,T4) : DT5 inchangé; File ← {(T4,T5), (T1, T2), (T4,T2), (T2, T1), (T5,T1)}

• (T4,T5) : DT4 inchangé; File ← { (T1, T2), (T4, T2), (T2, T1), (T5, T1)}

• (T1,T2) : DT1 inchangé; File ← { (T4, T2), (T2, T1), (T5, T1)}

• (T4,T2) : <u>DT4 = 4</u>; File ← {(T2, T1), (T5, T1), (T5, T4) }

• (T2, T1) : DT2 inchangé; File ← { (T5, T1), (T5, T4) }

• (T5,T1) : <u>DT5 = 2</u>; File ← { (T5, T4), (T4,T5)}

• Traitement de (T5, T4), (T4,T5) : DT5, DT4 inchangés,

==> File vide, AC-3 s'arrête.

# AC-3 : analyse

Nb d'arcs testés : 19

**Complexité** : temps : $O(a^3e)$
*(au pire 2e arcs seront rajoutés dans la file, et reduit-domaine examine $a^2$ paires d'etiquettes)*

**Pas encore optimal** : il y a encore des arcs explorés qui ne mènent à aucune réduction de domaine.

**Quand doit-on effectivement mettre à jour** le domaine d'une variable contrainte ?
Si une valeur dans le domaine d'une variable ne reçoit pas de support d'au moins une des autres variables auxquelles elle est liée.

**Autres optimisations de l'algorithme AC3** :

- <u>AC4</u> : complexité en $O(a^2e)$, mais complexité spatiale importante (Mohr R. et Henderson C., Artificial Intelligence, 28, 225-233)
- Version générique AC5 (1992), ...

# TP

- Soit le problèmes des n-reines: appliquez l'algorithme de retour arrière jusqu'à l'obtention de la première solution.
- Implanter AC-3

# Exercice 1

- Soit le problèmes des n-reines: il faut placer n reines sur un échiquier nXn afin que deux reines quelconques ne soient pas en prise. Proposez 2 PSC équivalents pour ce problème. Pour chacune de ces représentations, calculez le nombre de noeuds de l'arbre de recherche, le nombre de feuilles et la proportion nb de solutions / nb de feuilles.

# AC-4

- S'appuie sur la notion de *supports*
- Une valeur v d'une variable x est supportée s'il existe une valeur compatible dans les domaines des autres variables
- Si v est enlevée de Dx, on ne va pas systématiquement vérifier toutes les contraintes Cxy
  - On peut ignorer les valeurs de Dy qui ne s'appuient pas sur v comme support (cas où toutes les valeurs de Dy sont compatibles avec d'autres valeurs de Dx)
- Comme on doit gérer des compteurs de support pour chaque valeur de chaque domaine, AC-4 est très coûteux en mémoire
- Mais AC-4 est un algorithme optimal en temps pour la cohérence par arc.

# Directional Arc-Consistency

- Plus faible que les AC, mais plus rapide et parfois payant (il a été montré que sur certains problèmes « réels », il est possible de ne pas faire de retour arrière pendant la recherche simplement en garantissant la DAC).
- Idée : ne traiter les contraintes que selon un ordre total sur les variables
  - Si on traite $C_{X,Y}$, on ne traitera pas $C_{Y,X}$
- En pratique, les algorithmes utilisent souvent des ordres pendant la recherche… Souvent, l'ordre de DAC correspond
  - A l'ordre dans la recherche
  - A une structure, connue, ou trouvée en amont, du problème.

# Cohérences locales : introduction

Le simple problème de satisfiabilité est ***NP-complet*** :
on va définir des propriétés moins fortes que la cohérence.

**Principe** : on limite la propriété de cohérence à des sous-PSC de taille k
(propriété de ***cohérence locale***)
==>> à partir d'un PSC donné, on construit un PSC équivalent et localement consistant;
on infère localement et massivement des contraintes induites par le PSC et on les ajoute au PSC tant qu'elles ne sont pas déjà explicites (on parle aussi d'algorithmes de filtrage)

Le nouveau PSC est alors plus simple à résoudre par un algorithme de recherche du type backtrack.

**Algorithmes de filtrage** :

- peuvent être interrompus à tout moment : le PSC obtenu ne vérifie peut-être pas encore la propriété de locale cohérence cherchée, mais est plus simple
- sont incrémentaux pour l'ajout de contraintes

Dans la suite, on se limite à des ***contraintes binaires*** : la plupart des propriétés et algorithmes s'étendent au cas de contraintes n-aires, mais sont plus compliqués

# Cohérence par chemin

Propriété encore plus forte :

Un ***chemin*** $\{X_0, X_1, \ldots, X_m\}$ est ***cohérent*** si pour toute 2-étiquette composée

$\{<X_0, v_0>, <X_m, v_m>\}$ qui satisfait les contraintes (unaires et binaires) sur $X_0$ et $X_m$, il existe une étiquette pour chacune des variables $X_1$ à $X_{m-1}$ telle que toutes les contraintes (unaires et binaires) sur les variables adjacentes sont satisfaites.

Un PSC P (V,D,C) est ***cohérent par chemin*** si tout chemin du graphe de contraintes associé à P est cohérent.

**Propriétés** :

- un PSC P est cohérent par chemin si tous les chemins de longueur 2 du graphe de contraintes associé à P sont cohérents (Montanari, 1974).
- pour les PSC binaires, cohérence par chemin = 3-cohérence (Freuder, 1982)
- n'est pas une condition nécessaire (ni suffisante) pour la satisfiabilité

# Exemple

PSC = (V, D, C):
V = (X, Y, Z)
D = ($D_X$ , $D_Y$ , $D_Z$) avec $D_X = D_Y = D_Z = [0..10]$
C: (X+Y = 10; Y+Z < 10)

- **Satisfiable** : oui

- **Cohérence par arc**

On fait tourner AC3 : au départ, File = { ( X, Y), (Y, Z), (Y, X), (Z, Y) }
on obtient à la fin :
$D_X \in [1..10]$; $D_Y$, $D_Z \in [0..9]$

- **Cohérence par chemin**

<X,1>,<Z,0> ok Y=9 satisfait les deux contraintes de C
<X,1>,<Z,1> pas d'assignation de Y qui satisfait les deux contraintes
<X,1>,<Z,2> idem

En fait, aucune des étiquettes composées $<X,v_X>,<Y,v_Y>,<Z,v_Z>$ telles que $v_Z \geq v_X$ ne satisfait P (car C1 => Y = 10 - X et C2 => 10 - X + Z < 10)

# k- Cohérence

Un PSC P = (V,D,C) est ***1-cohérent*** ssi pour toute variable X, il existe une valeur v dans $D_X$ tq <X, v> satisfait les contraintes pertinentes sur X.

Un PSC est ***k-cohérent*** pour k>1 ssi toutes les k-1 étiquettes composées qui satisfont les contraintes pertinentes de C peuvent être étendues pour inclure toute k-ième nouvelle variable afin de former une k-étiquette composée qui satisfasse les contraintes pertinentes (i.e. portant sur ces k variables).

**Exemple** : PSC 3-cohérent et non 2-cohérent
V = { X, Y, Z }
$D_X = D_Z$ = { rouge } , $D_Y$ = { rouge, bleu }
C = { C1, C2 } ; C1 : X ≠ Y ; C2 : Y ≠ Z

• non 2-cohérence : l'étiquette <Y, rouge > ne peut pas être étendue en une 2-étiquette comprenant X sans violer la contrainte C1

• 3-cohérence : 2-étiquettes satisfaisant les contraintes de C
(<X, rouge>, <Y, bleu >), (<X, rouge>, <Z, rouge>) et (<Y, bleu>, <Z, rouge>) qui peuvent être étendues en la 3-étiquette satisfaisant C
(<X, rouge>, <Y, bleu >, <Z, rouge>)

# Propriétés

Un PSC est ***fortement k-cohérent*** s'il est k'-cohérent pour tout k',1 ≤ k' ≤ k

**Propriété** : Soit |V|= n : n-cohérence forte et 1-Satisfiabilité
-> satisfiabilité

**Exemple** : PSC de 4 variables 3-cohérent mais non satisfiable
(problème de coloriage de cartes)

V = { X, Y, Z, U}
$D_X = D_Y = D_Z = D_U$ = { rouge, vert, bleu }
C : C1 = X ≠ Y; C2 : X ≠ Z ; C3 : X ≠ U ; C4 : Y ≠ Z ; C5 : Y ≠ U ; C6 : Z ≠ U

- 3-cohérence : avec 2 variables assignées, on peut toujours assigner la 3ème couleur à la 3 ème variable, quelle qu'elle soit
- insatisfiabilité : il faudrait 4 couleurs ...

**Propriétés** :
- Cohérence par noeud = 1-cohérence
- Cohérence par arc = 2-cohérence
- Cohérence par chemin = 3-cohérence (uniquement pour les PSC binaires)

Ex de PSC cohérent par arc et non satisfiable:
X≠Y, X ≠ Z, Y ≠ Z, $D_X = D_Y = D_Z$ = {1, 2 }

# Résumé des techniques de réduction de problèmes

**Cohérence par chemin**: complexité spatiale et temporelle assez lourde; intéressante dans certain types de problèmes pour lesquels la 3-cohérence forte assure la satisfiabilité globale ; expérimentalement jugée lourde

| Techniques | Complexité en temps | Complexité Espace |
|---|---|---|
| Cohérence par noeud | $O(an)$ | $O(an)$ |
| Cohérence par arc | $O(a^2e)$ | $O(a^2e)$ |
| Cohérence par chemin | $O(a^3n^3)$ (algorithme PC3) | $O(a^3n^3)$ |

- Compromis tps de réduction - tps de recherche
- Techniques de réduction à utiliser avant la recherche ou en couplage avec la recherche

# Partie II

Algorithmique avancée
pour les PSC

# Mieux équilibrer la recherche

Jusque là :

1. Réduction du problème
2. Résolution par des méthodes « brutales »

L'idée est de réduire le problème
pendant la recherche

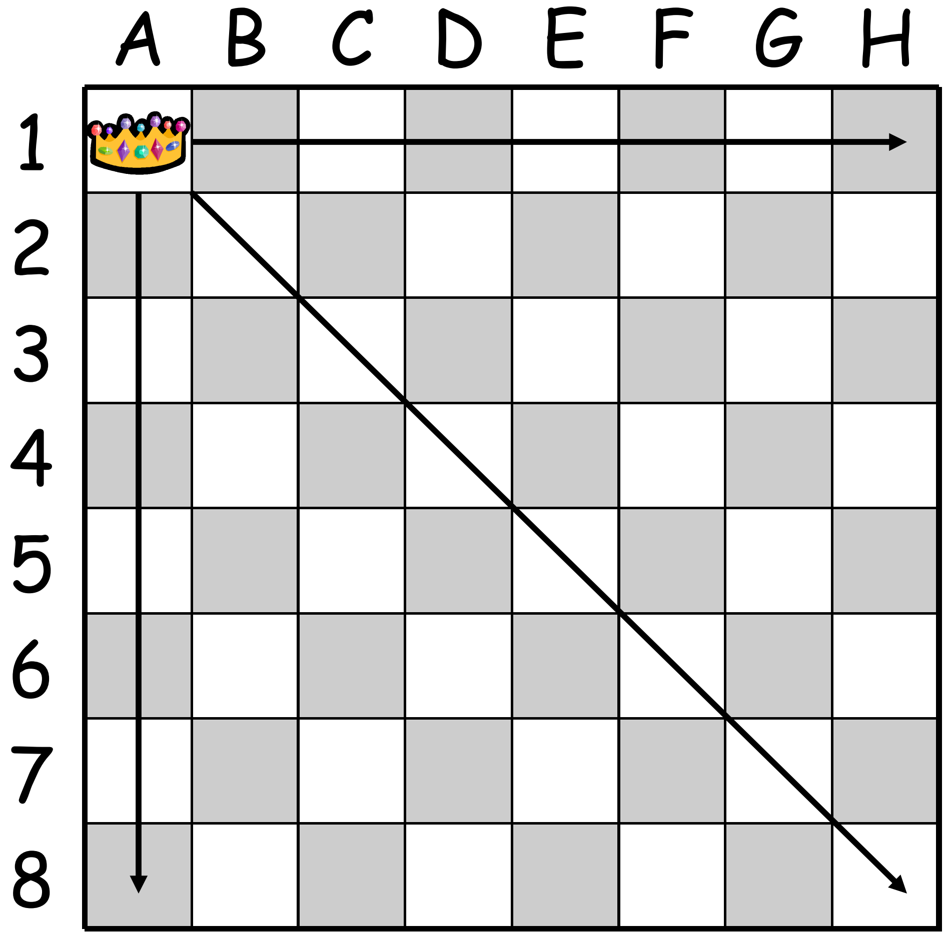
Méthodes de recherche plus élaborée

On retrouve toujours le compromis

« Essayer ou réfléchir »

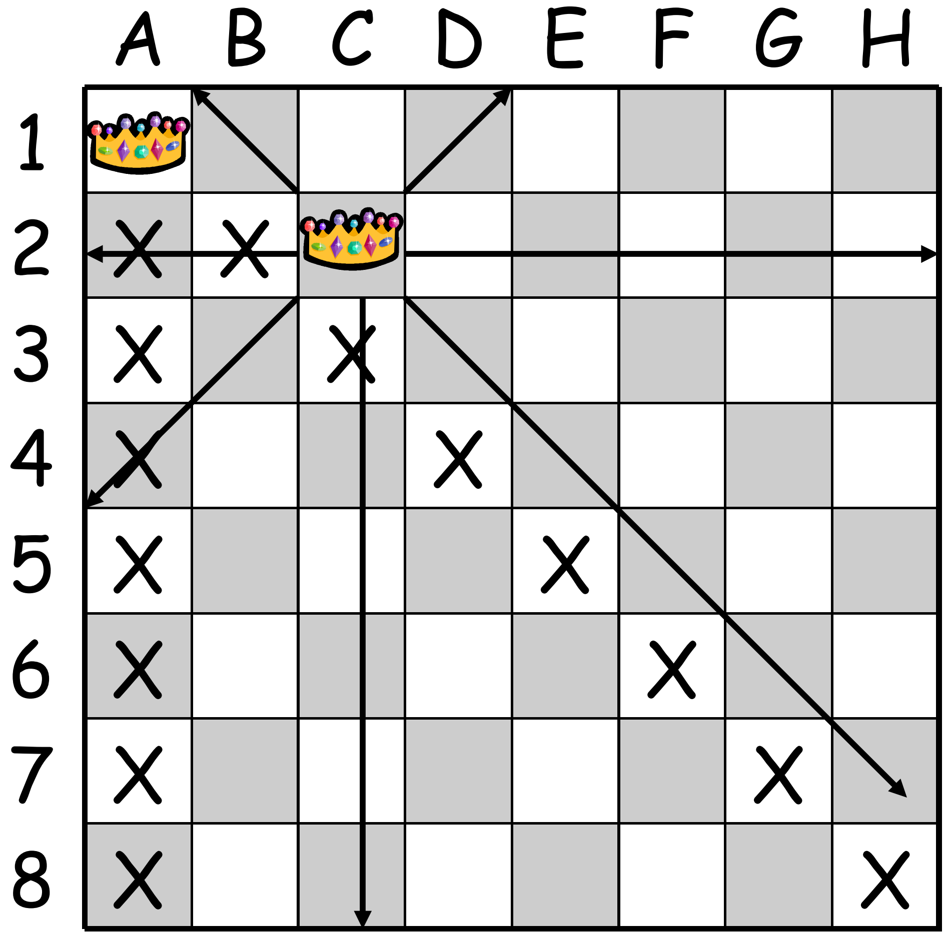
# Stratégies de lookahead

Repérer en avance les branches vouées à l'échec



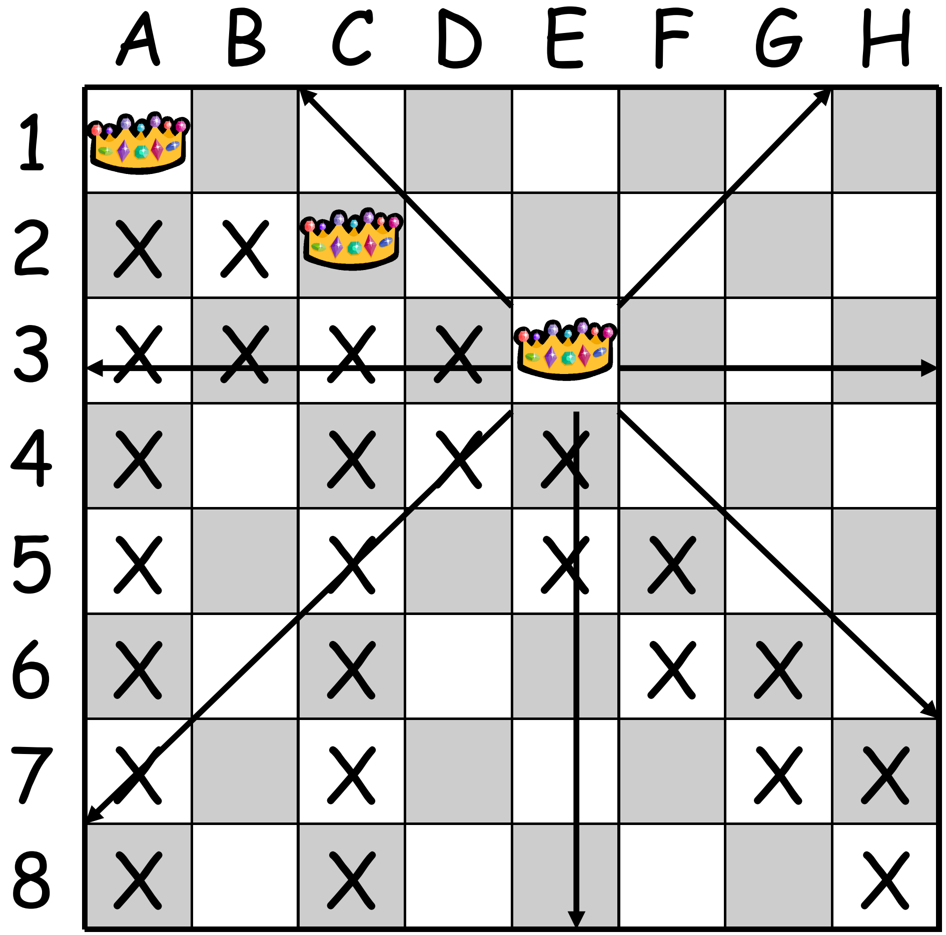
# Stratégies de lookahead

Repérer en avance les branches vouées à l'échec



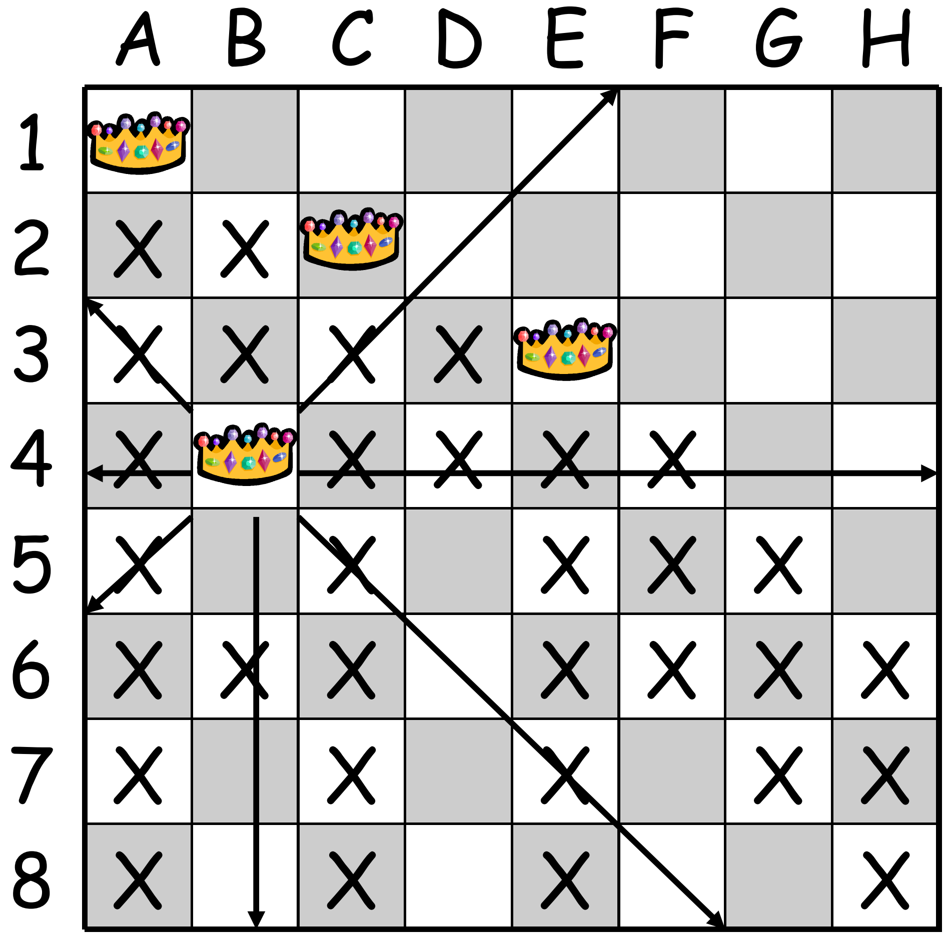
# Stratégies de lookahead

Repérer en avance les branches vouées à l'échec



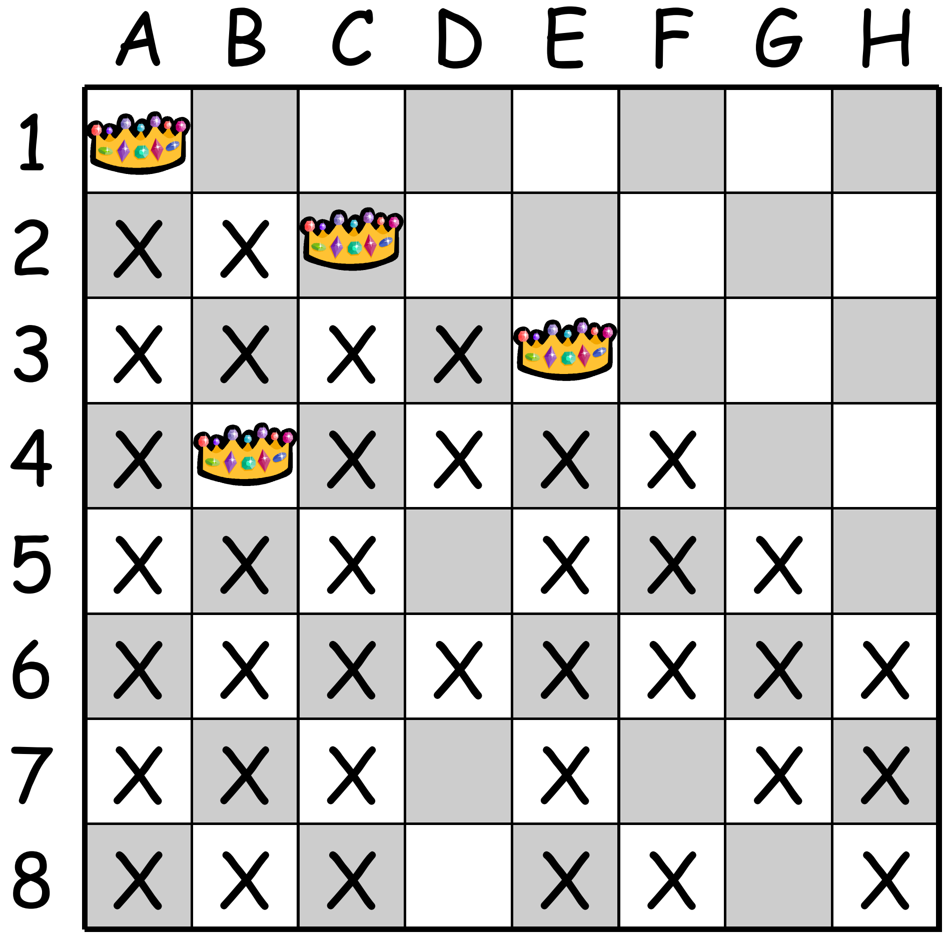
# Stratégies de lookahead

Repérer en avance les branches vouées à l'échec



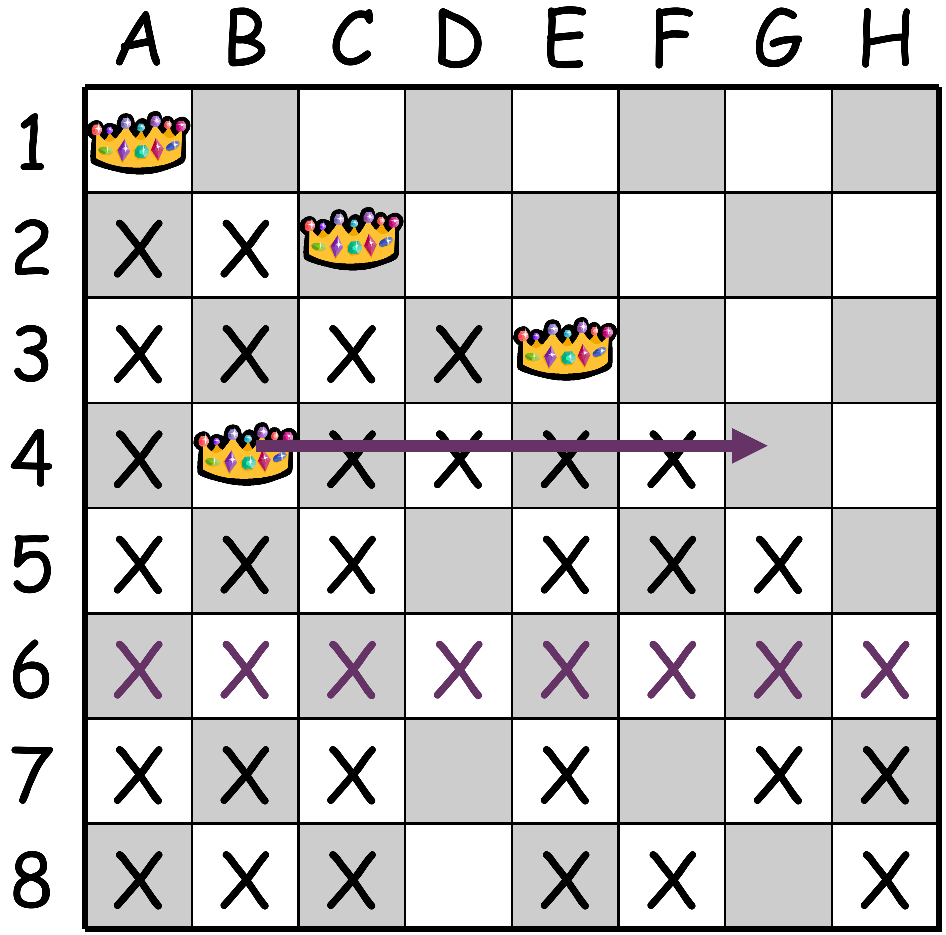
# Stratégies de lookahead

Repérer en avance les branches vouées à l'échec



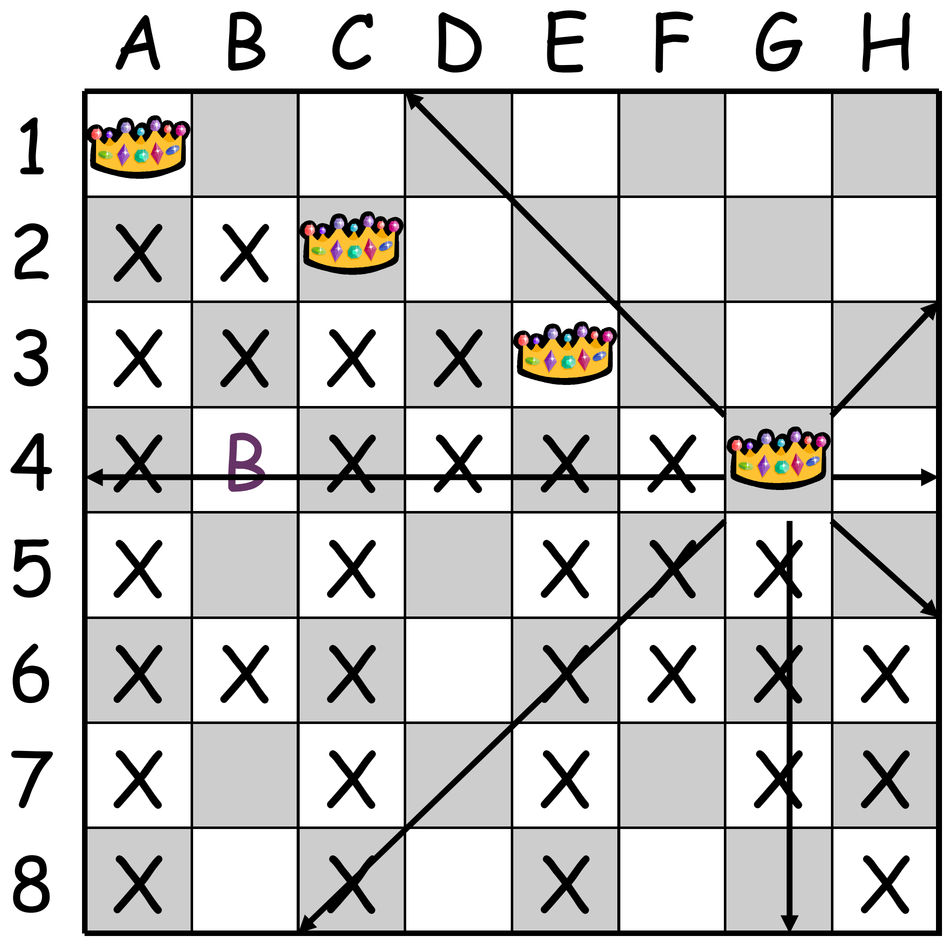
# Stratégies de lookahead

Repérer en avance les branches vouées à l'échec



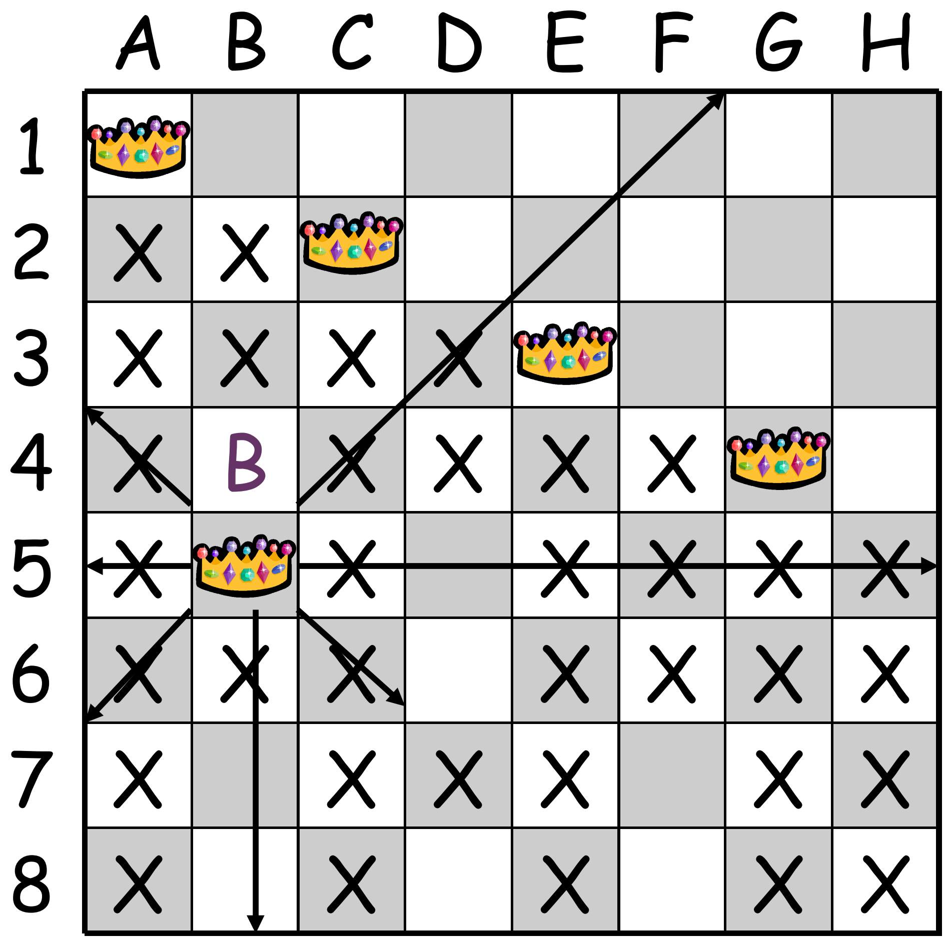
# Stratégies de lookahead

Repérer en avance les branches vouées à l' échec



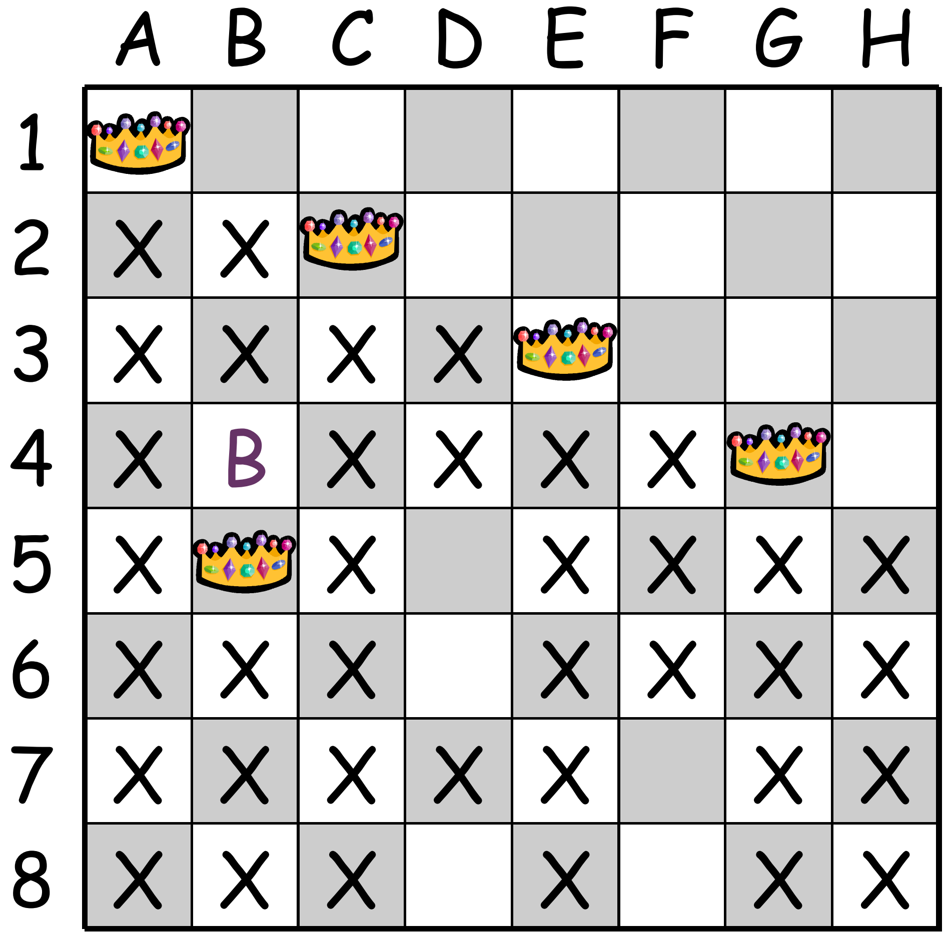
# Stratégies de lookahead

Repérer en avance les branches vouées à l'échec



# Stratégies de lookahead

Repérer en avance les branches vouées à l’échec



# Stratégies de lookahead (suite)

**Algorithme classique de Backtrack**

**Procédure retour_arrière(V, D, C)**
**Res ← backtrack(V, ∅ , D, C);**
**return(Res)**

**Procédure backtrack(NE, EC, D, C)**
**Si NE = ∅ alors return(EC)**
**sinon**
    **Sélectionner une variable X dans NE;**
    **Répéter**
        **Sélectionner une valeur v dans $D_x$; $D_x \leftarrow D_x - \{ v \}$;**
        **si EC ∪ <X, v> ne viole pas les contraintes de C**
            **alors**
            **Résultat← backtrack( NE - { X }, EC ∪ <X, v>, D, C)**
            **si Résultat ≠ ∅ alors return(Résultat)**
        **finsi**
    **jusqu'à ce que $D_x = \emptyset$**
**return( ∅ )**
**finsi**

# Stratégies de lookahead (suite)

**Procédure maj-1(NE,D,C, E)**
**Pour tout Y $\in$ NE**
**Pour tout v $\in$ $D_y$**
**si <Y, v > (avec E) ne satisfait plus $C_y$**
**alors Dy $\leftarrow$ Dy - { v }**
**finpour**
**finpour**
**retourner(D)**

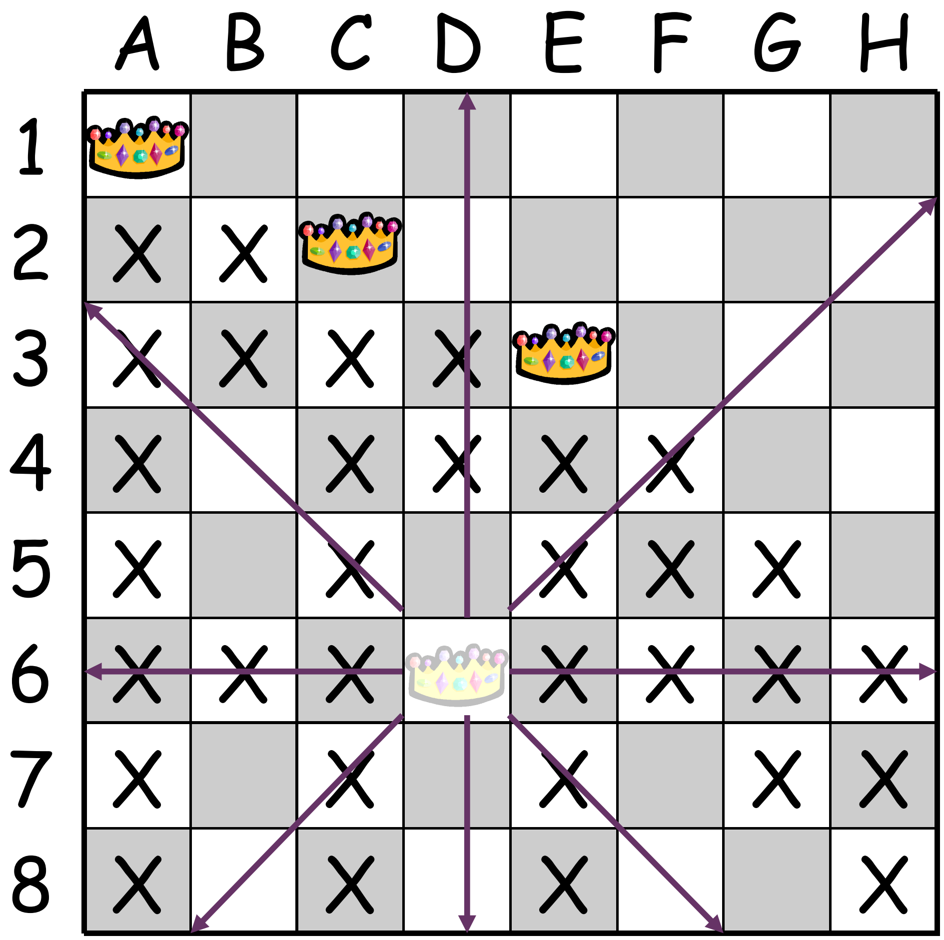
**Procédure FC(V, D, C)**
**Res $\leftarrow$ ForwardChecking(V, $\varnothing$ , D, C);**
**return(Res)**

**Procédure ForwardChecking(NE, EC, D, C)**
**Si NE = $\varnothing$ alors return(EC)**
**sinon**
**Sélectionner une variable X dans NE;**
**Répéter**
**Sélectionner une valeur v dans $D_x$; $D_x \leftarrow D_x$ - { v };**
**si EC $\cup$ <X, v> ne viole pas les contraintes de C**
**alors**
**D' $\leftarrow$ maj-1(NE - { X }, D, C, <X, v>)**
**Si aucun domaine de D' n'est vide alors**
**Résultat $\leftarrow$ ForwardChecking( NE - { X }, EC $\cup$ <X, v>, D', C)**
**si Résultat $\neq$ $\varnothing$ alors return(Résultat)**
**finsi**
**jusqu'à ce que $D_x = \varnothing$**
**return( $\varnothing$ )**

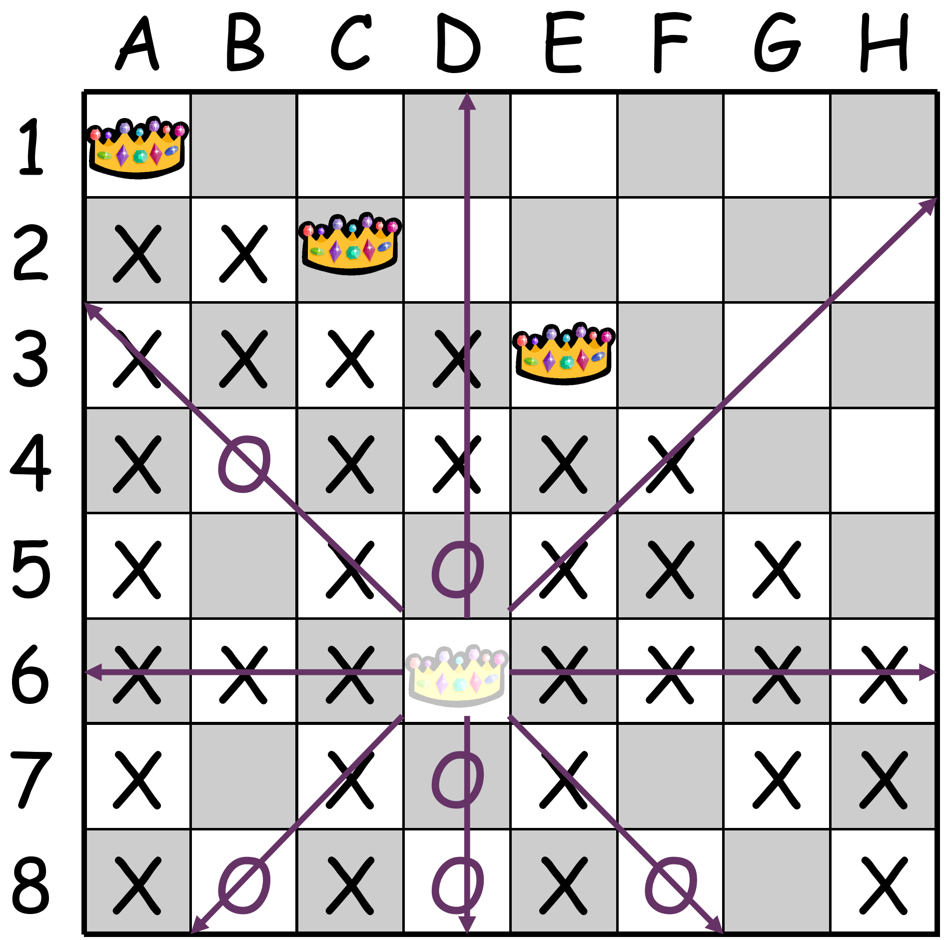
# Peut-on encore passer plus de temps (mais pas trop) à chaque noeud ?

- FC *ressemble* à un maintient de la cohérence par nœud à chaque étape du calcul, en plus complet… (attention, ça ressemble, mais ce n'en est pas… FC ne maintient la cohérence que pour les valeurs non encore instanciées)
- On doit pouvoir aller plus loin en essayant de maintenir la cohérence par arc par exemple…
- Algos de type MAC (Maintaining Arc Consistency)
- Comment trouver la limite entre « essayer » et « réfléchir » ?
  - Quelle différence en pratique, puisqu'au niveau algorithmique, « réfléchir » revient aussi à « essayer »… ?

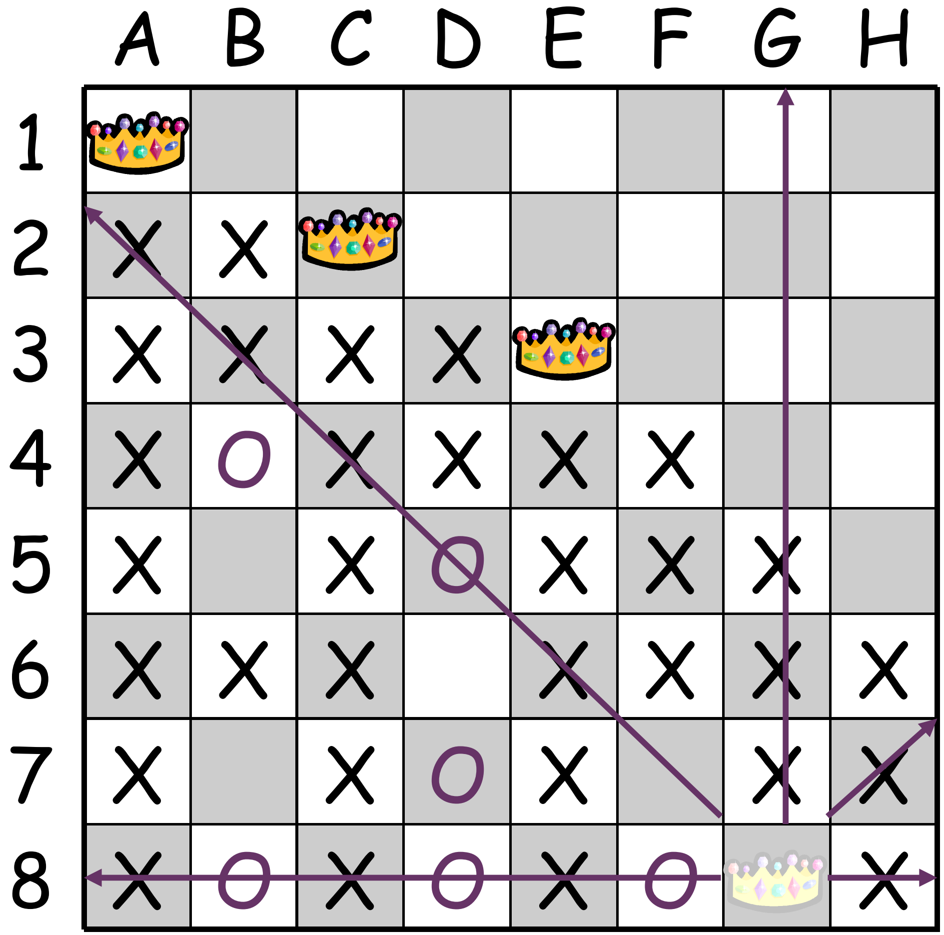
# Stratégies de lookahead
# full lookahead



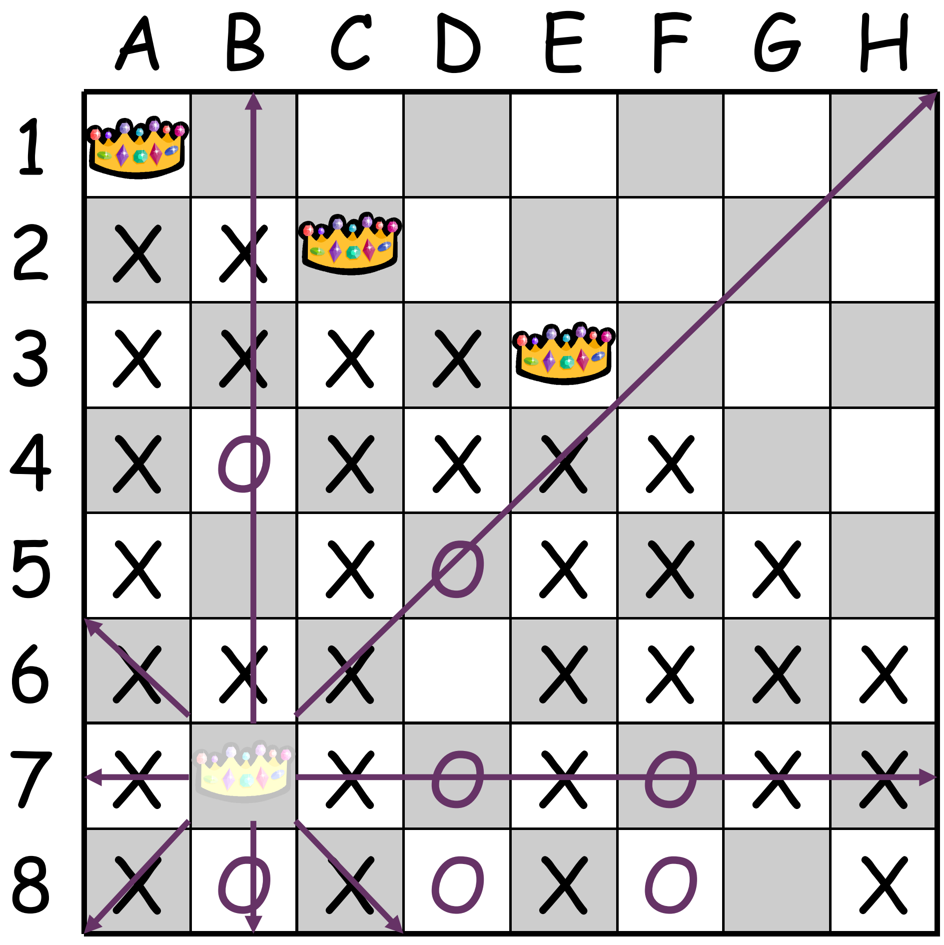
# Stratégies de lookahead
## full lookahead



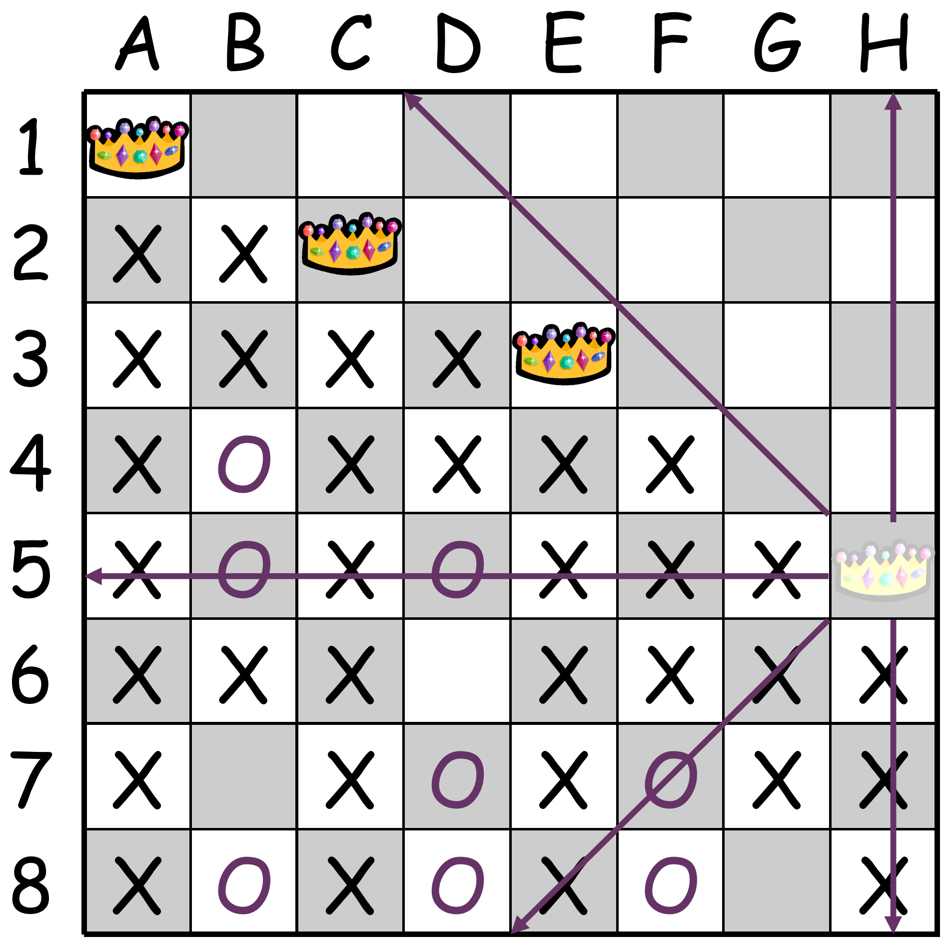
# Stratégies de lookahead
## full lookahead



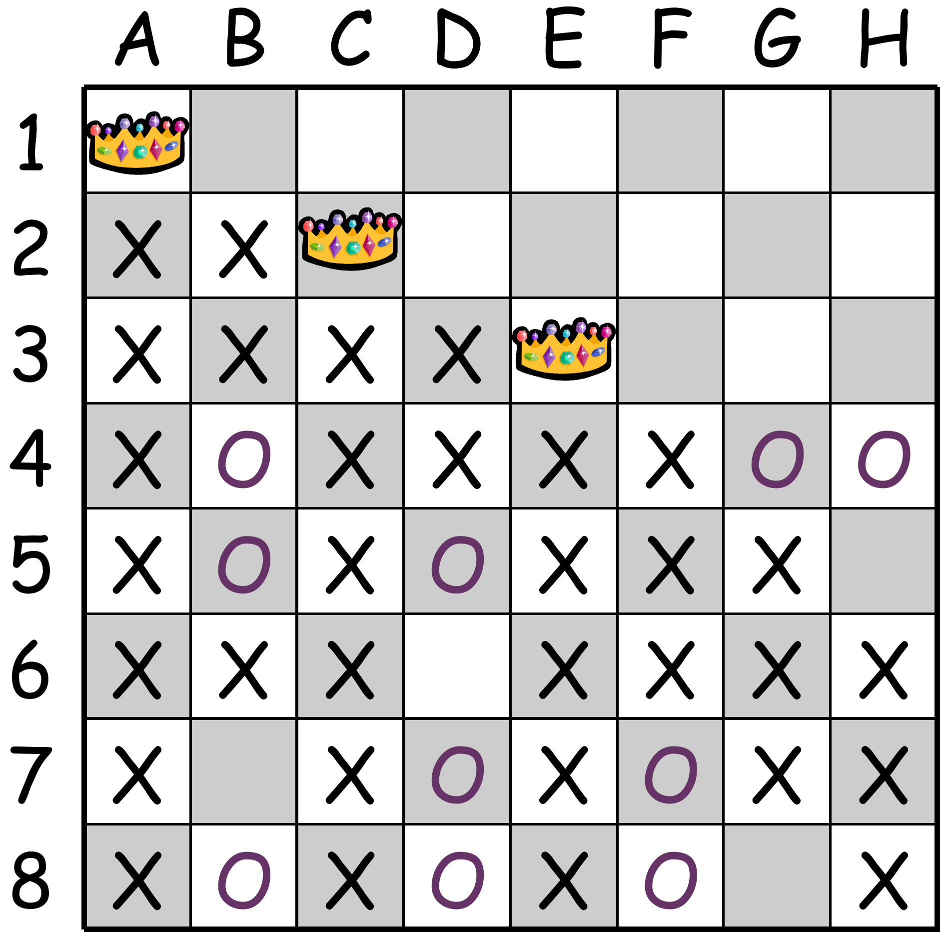
# Stratégies de lookahead
# full lookahead



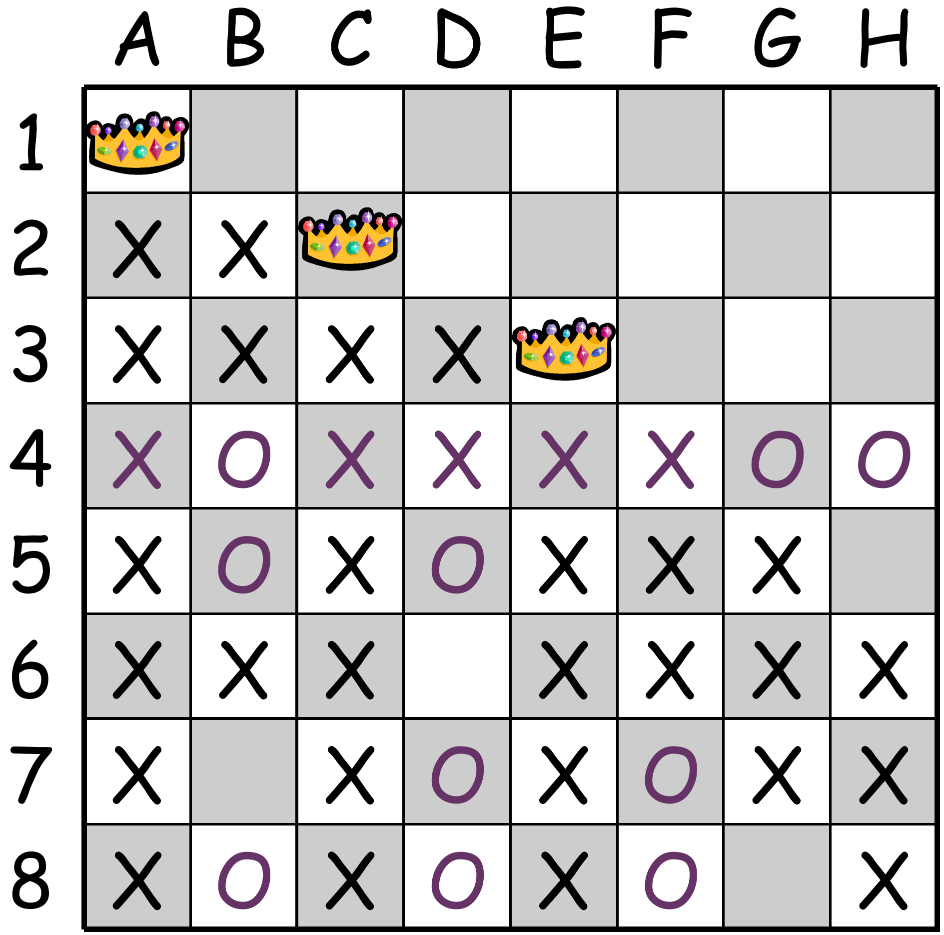
# Stratégies de lookahead
## full lookahead



# Stratégies de lookahead
## full lookahead



# Stratégies de lookahead
## full lookahead



# full lookahead

```
Procédure FullLookahead(V, D, C)
Res ← AC-Lookahead, ∅ , D, C);
return(Res)

Procédure AC-Lookahead(NE, EC, D, C)
Si NE = ∅   alors return(EC)
sinon
     Sélectionner une variable X dans NE;
     Répéter
          Sélectionner une valeur  v dans D_x; D_x←        D_x - { v };
          si EC ∪  <X, v> ne viole pas les contraintes de C
               alors
               D' ←    maj-1(NE - { X }, D, C, <X, v>)
                                   …
               Si aucun domaine de D' n'est vide alors
                    Résultat← backtrack( NE - { X }, EC ü  ü   <X, v>, D', C)
                    si Résultat ≠      ∅  alors return(Résultat)
               finsi
          finsi
     jusqu'à ce que D_x = ∅
return( ∅  )
finsi
```

# full lookahead

**Procédure FullLookahead(V, D, C)**
**Res ← AC-Lookahead, ∅ , D, C);**
**return(Res)**

**Procédure AC-Lookahead(NE, EC, D, C)**
**Si NE = ∅ alors return(EC)**
**sinon**
**Sélectionner une variable X dans NE;**
**Répéter**
**Sélectionner une valeur v dans $D_x$; $D_x$← $D_x$ - { v };**
**si EC ∪ <X, v> ne viole pas les contraintes de C**
**alors**
**D' ← maj-1(NE - { X }, D, C, <X, v>)**
**(NE-{x}, D'', C) ← AC-X(NE-{x}, D', C)**
**Si aucun domaine de D'' n'est vide alors**
**Résultat← AC-Lookahead ( NE - { X }, EC ü ü <X, v>, D', C)**
**si Résultat ≠ ∅ alors return(Résultat)**
**finsi**
**finsi**
**jusqu'à ce que $D_x$ = ∅**
**return( ∅ )**

# Full lookahead ?

Le nom « **full lookahead** »
peut être trompeur

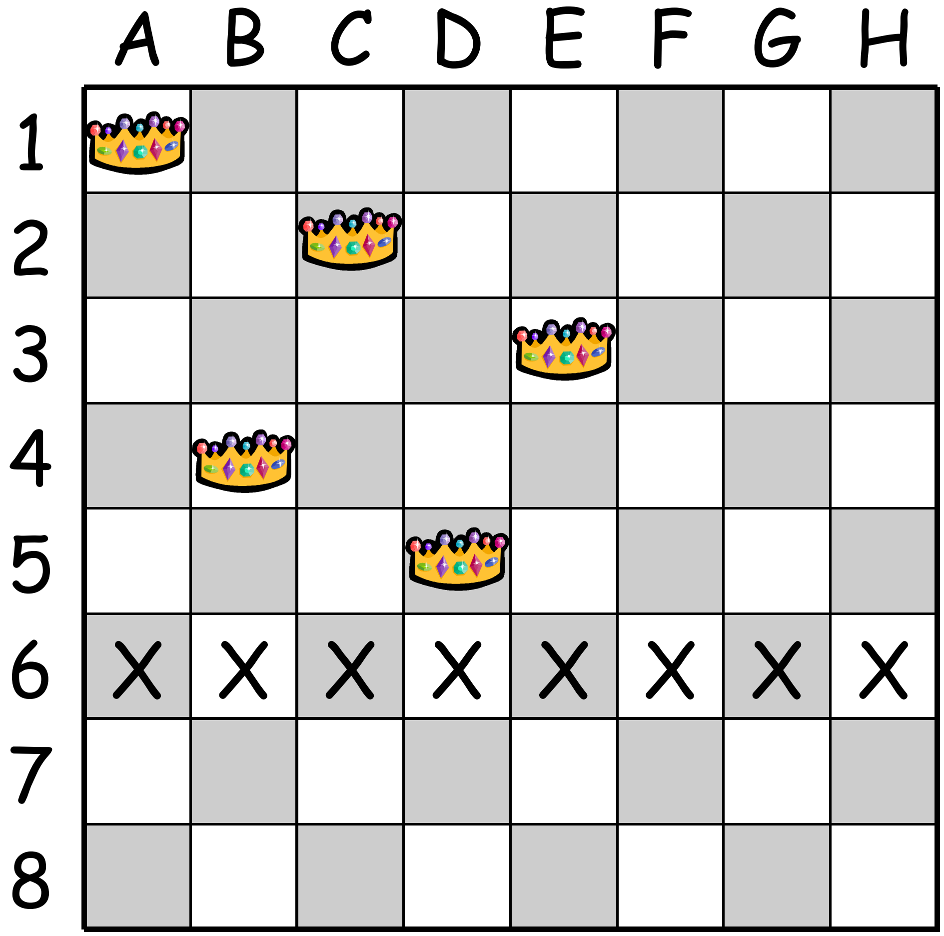
- D'autres propriétés, encore plus fortes, peuvent être maintenue…
  - Cohérence de chemin
  - K-cohérence
  - …

En pratique pourtant, ce choix est un bon compromis…

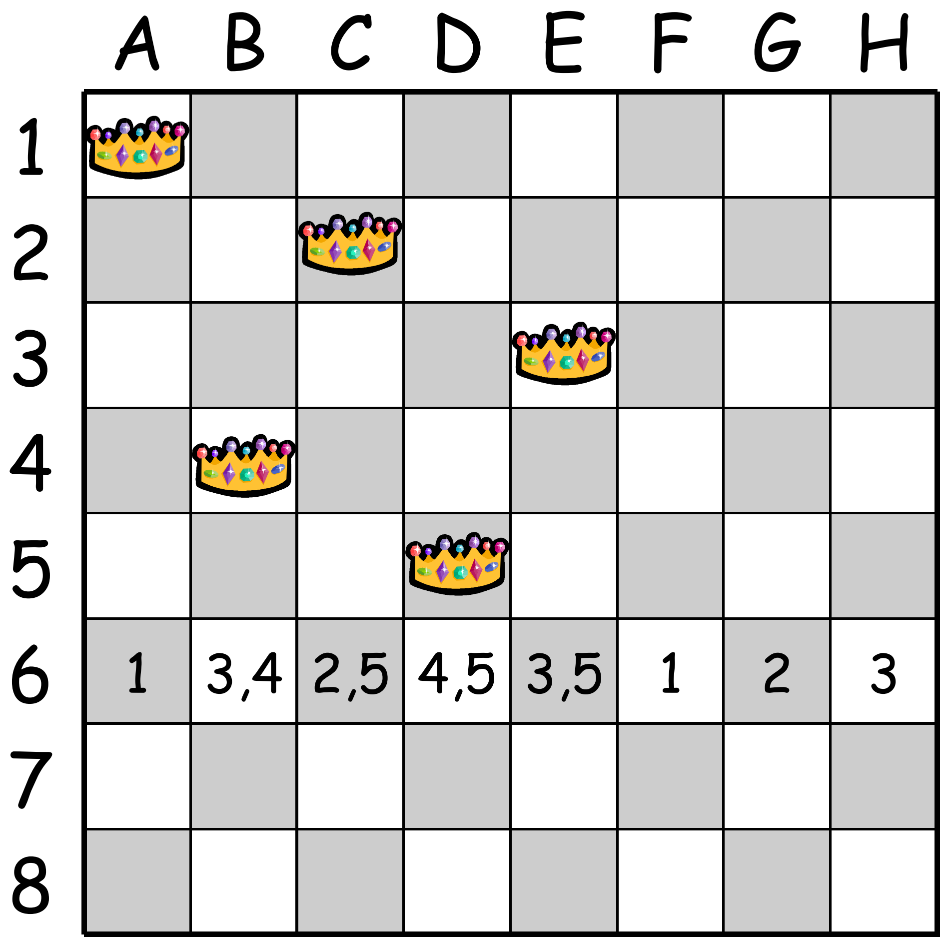
# Où peut-on encore améliorer les performances ?

- Nous sommes partis d'un retour arrière classique et avons essayer de limiter les retours arrières…
- Pour cela, on a utilisé des techniques pour « explorer » l'espace de recherche avant d' aller vraiment (lookahead & full lookahead)
- L'idée est maintenant d'exploiter au maximum le travail fait, si jamais celui-ci a par exemple conduit à un échec…
  de lookahead, on passe au « lookback »…
- Le but est maintenant de passer du temps à analyser la situation d'échec pour en déduire des informations

# Ne pas retourner dans une recherche vouée à l’échec

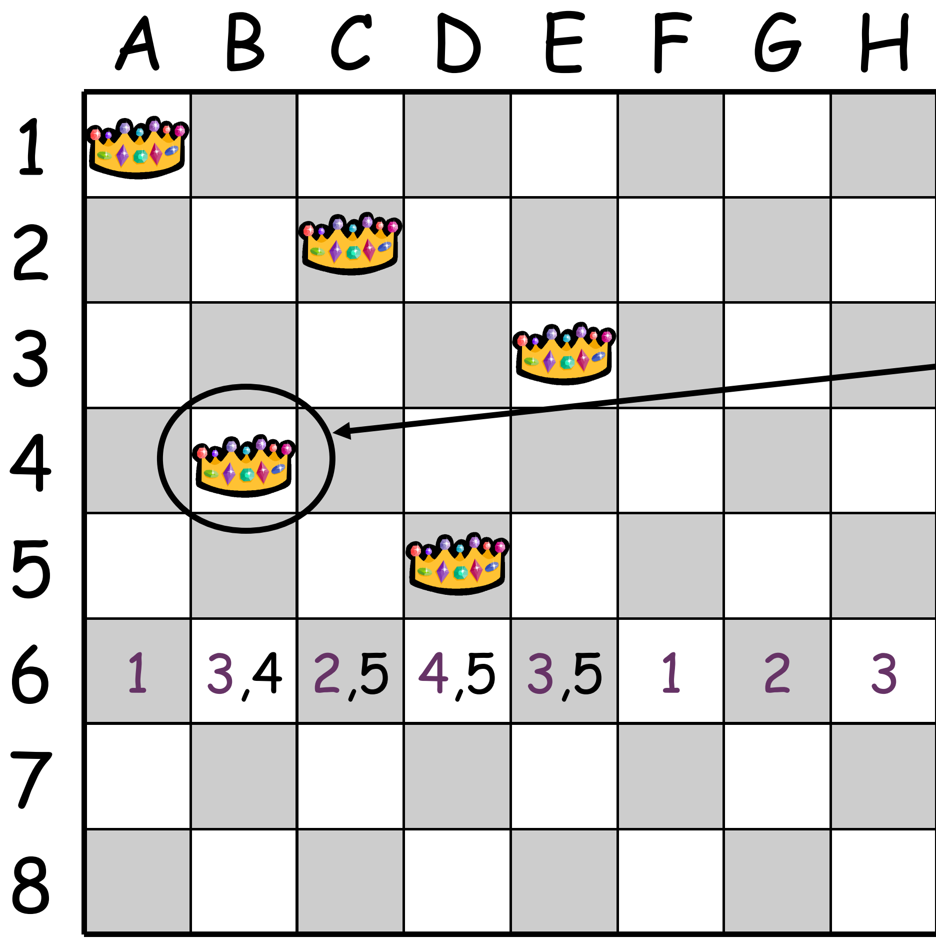


# Ne pas retourner directement dans une recherche vouée à l’échec




**Pour chaque case, on identifie les plus anciens responsables de l’échec**

# Ne pas retourner directement dans une recherche vouée à l’échec




**Il faut remonter directement au choix de cette reine**

**Pour chaque case, on identifie les plus anciens responsables de l’échec**

# Retour arrière « intelligents » Backjumping

- Dans l'algo, il faut pouvoir remonter rapidement à un précédent point de choix
- Il faut pouvoir identifier les points de choix pertinents
  - Il faut mémoriser le niveau dans l'arbre de recherche,
  - Étiqueter chaque valeur non valide par le niveau L minimum où la valeur a été invalidée (premier niveau empêchant la valeur)
  - Pour trouver le niveau où remonter, il faut prendre le max des L pour toutes les étiquettes possibles
  - Si plusieurs variables ont un domaine vide, on peut prendre le min de ces variables…
    - Cela donne le min du max des min…

# Version simplifiée du Backjumping

Parfois, la recherche du précédent point de choix peut être couteuse… On veut pouvoir faire du backjump mais sans que cela coûte aussi cher, quitte à avoir quelque chose de moins efficace

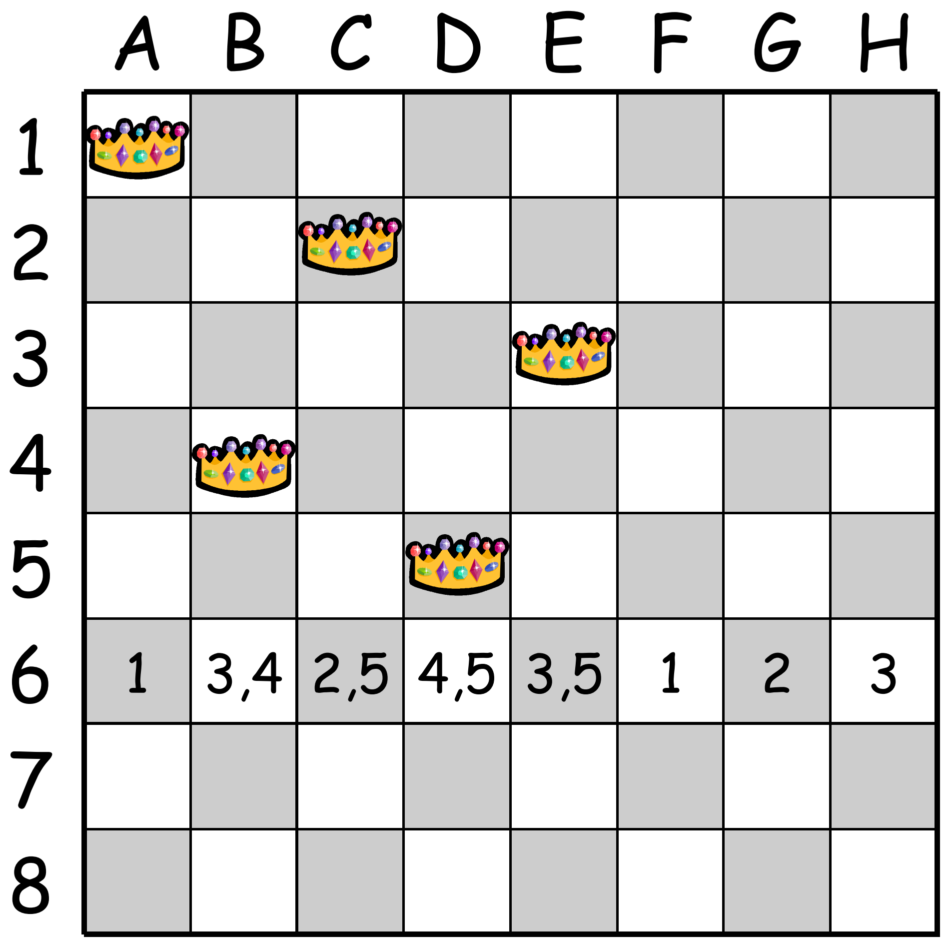
Et puis: Redondant avec Forward Checking…

## Graph-based BackJumping

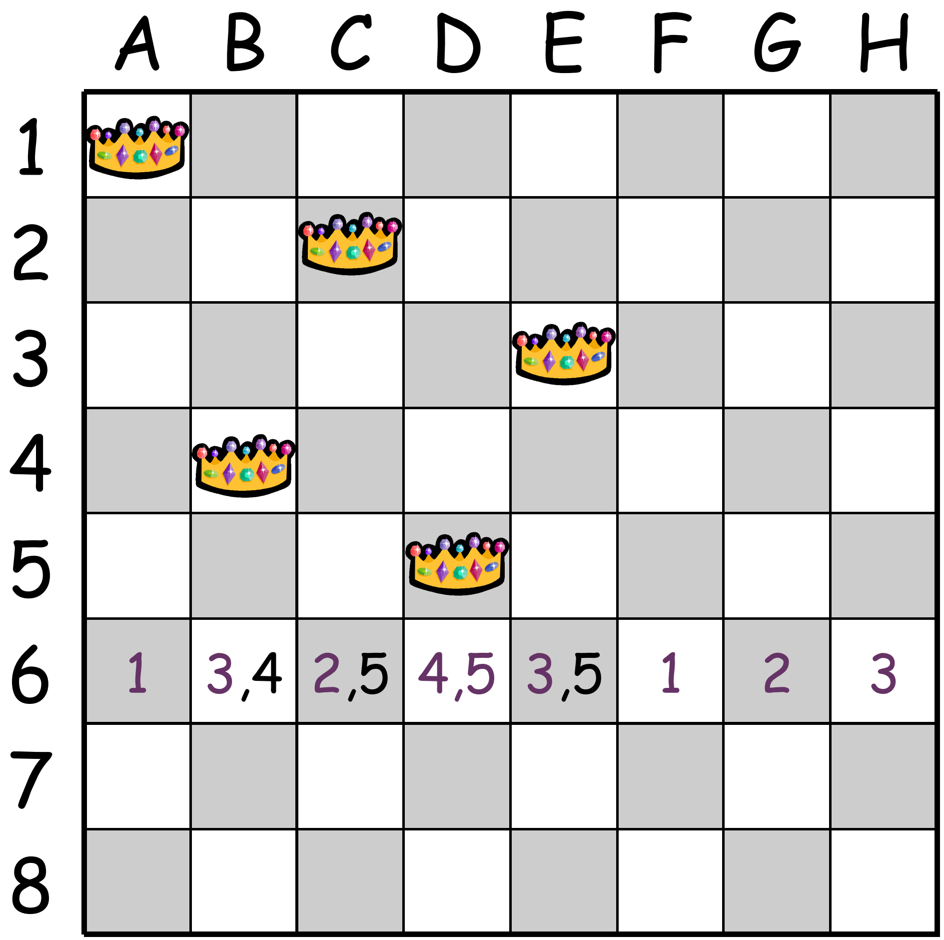
L'idée est d'utiliser le graphe des contraintes pour ne remonter qu'aux choix qui ont un lien avec l'échec courant.

L'analyse du graphe est plus simple que le calcul du niveau de retour dans le backjump classique.

# Apprendre de l’échec



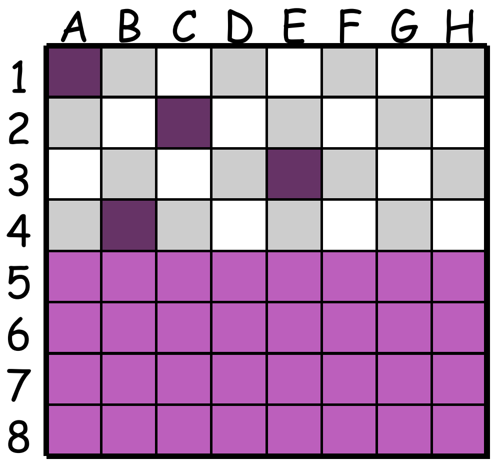
# Apprendre de l’échec

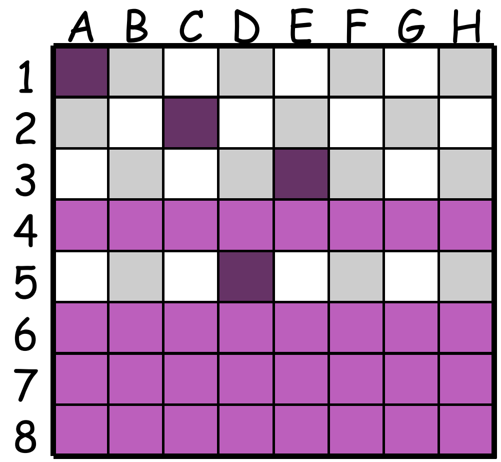



Explications minimales de l’échec :
{ 1A, 2C, 3E, 4B} et { 1A, 2C, 3E, 5D}

# Apprendre de l' échec

Explications minimales de l'échec :
{ 1A, 2C, 3E, 4B} et { 1A, 2C, 3E, 5D}




**Quelques soient les valeurs des autres reines, ces configurations ne doivent plus jamais être explorées dans les recherches futures**

# Apprentissage en cours de recherche Nogoods

- Les explications minimales des échecs sont appelées « nogoods » dans la littérature
- Certains algorithmes ne se focalisent pratiquement que sur la découverte la plus rapide du maximum de nogoods courts
- Par contre, trouver tous les nogoods et les garder en mémoire n'est pas possible en pratique
  - Il y a beaucoup trop de nogoods à récolter pendant une recherche
  - Minimiser les explications est un gros problème
- Les algorithmes se focalisent sur la découverte de certaines explications (courtes, …) quitte à passer à côté d'autres explications...

# Comment choisir la prochaine variable ?

- L'ordre de choix des variables à affecter est crucial
  - Impact direct sur la taille de l'arbre de recherche
  - Pour les problèmes de satisfiabilité des PSC, la différence de performance entre deux heuristique peut être exponentiellement grande (une branche vs tout l'arbre de recherche)
- Le choix des heuristiques dépend surtout du type de problème étudié
  - Redondance des contraintes,
  - Uniformité des contraintes (problèmes aléatoires, …)
- Introduction de hasard dans le choix ?

# Exemples d'heuristiques

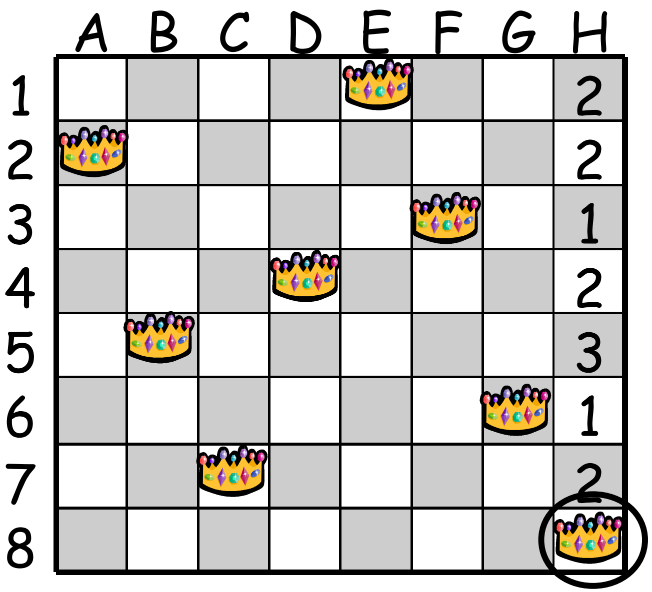
- Heuristiques statiques
  - Minimal Width Heuristic
    - Réduire le nombre $max$ de retours arrières
  - Minimal Bandwidth Heuristic
    - Réduire la hauteur des retours arrières

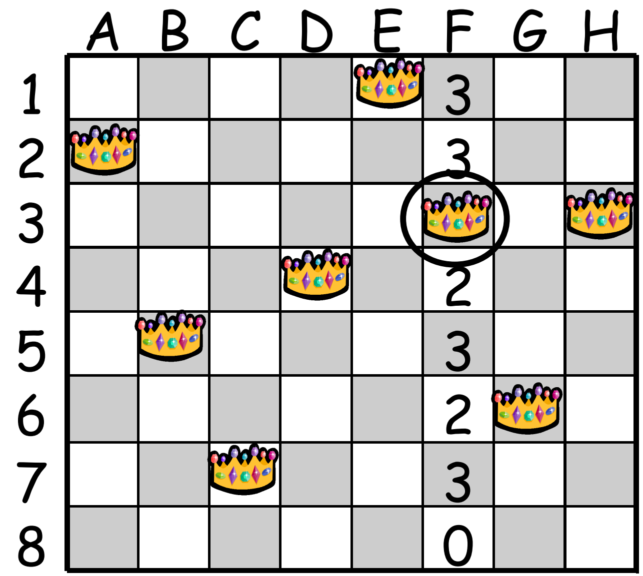
- Des heuristiques dynamiques, simples et efficaces
  - Choisir d'abord la tâche qui a le plus de chance d'échouer. Revient à identifier le plus rapidement possible les branches vouées à l'échec
  - Choisir la variable la plus contrainte d'abord
  - Choisir la variable qui entraîne le moins de conflits avec les autres variables
  - Choisir la variable qui satisfait les contraintes les plus souvent violées dans l'historique de la recherche
  - Restarts
  - …

# Recherches locales sur les CSP

- Partir d'une assignation initiale
- Changer une valeur à la fois pour arriver à la solution
- Heuristique : Choisir la variable qui va permettre de se rapprocher de la solution, avec tirages aléatoires
- Très efficace en pratique (N-Queens, Hubble, …)

# Recherche locale



# Petit sujet problématique

**Comment évaluer un algorithme**

**sur des PSC** ?