



Fondement de l'Intelligence Artificielle



Abir CHaabani

abir.chaabani@gmail.com

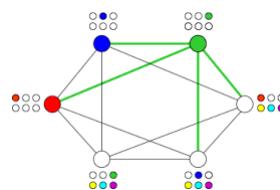
abir.chaabani@enicar.ucar.tn

Université de Carthage - ENICARTHAGE

2 GINF 2024/2025

1

Chapitre 3



Problèmes à satisfaction de contraintes

2

Plan Chapitre 3

- Rappel
- Description des CSP's
- Exploration avec backtracking
- Améliorations du backtracking
 - Most constrained variable
 - Most constraining variable
 - Least constraining value
 - Forward checking
 - Propagation des contraintes
 - Cohérence de arcs
- CSP local: Formulation avec min-conflicts.

3

Rappel

- ❖ Nous avons vu qu'un certain nombre de problèmes intéressants peuvent être résolus en les formulant comme des problèmes de recherche dans un graphe d'états :
 - ❖ On tient compte des aspects spécifiques à l'application en définissant une fonction heuristique (h) qui guide l'exploration efficace du problème;
 - ❖ La fonction de transition tient compte de l'aspect dynamique de l'application.
 - ❖ Par contre, les états (les noeuds) du graphe sont « opaques » vis-à-vis de la fonction de transitions :
 - Les successeurs ne dépendent de manière explicite de la structure interne de l'état.

4

Plan Chapitre 3

- Rappel
- Description des CSP's**
- Exploration avec backtracking
- Améliorations du backtracking
 - Most constrained variable
 - Most constraining variable
 - Least constraining value
 - Forward checking
 - Propagation des contraintes
 - Cohérence de arcs
- CSP local: Formulation avec min-conflicts.

5

Problème de satisfaction de contraintes

- En raisonnement par contraintes, un modèle est construit en utilisant
 - des **variables**
 - des **domaines**
 - des **contraintes**
- Un tel modèle est appelé **Problème de Satisfaction de contraintes (CSP)**



- Une solution à un CSP est: affecter à chaque variable une valeur de son domaine de manière à satisfaire toutes les contraintes.

6

Pourquoi CSP?

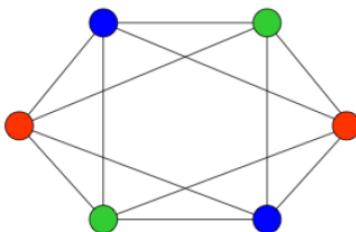
- La résolution de problèmes de satisfaction de contraintes peut être vue comme un cas particulier de la recherche heuristique
- La structure interne des états (nœuds) a une représentation particulière
 - ✓ un état est un ensemble **de variables** avec **des valeurs** correspondantes
 - ✓ les transitions entre les états tiennent compte **de contraintes** sur les valeurs possibles des variables
- Sachant cela, on va pouvoir utiliser **des heuristiques générales**, plutôt que des heuristiques spécifiques à une application
- En traduisant un problème sous forme de satisfaction de contraintes, on élimine la difficulté de définir l'heuristique $h(n)$ pour notre application

7

Exemple 1: Problème de coloriage de graphe

Coloriage des nœuds d'un graphe:

Quel est le nombre **minimal** de couleurs tel que deux nœuds adjacents reçoivent des couleurs différentes.

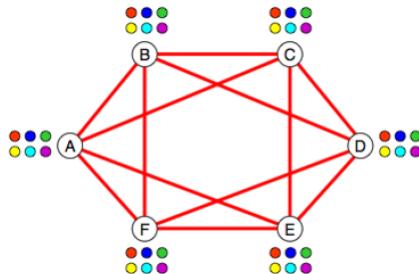


8

Exemple 1: Problème de coloriage de graphe

Modèle complet :

- Variables : A, B, C, D, E, F.
- Domaines: rouge bleu vert jaune bleu-ciel violet
- Contraintes : A ≠ B, A ≠ C, A ≠ E, A ≠ F, B ≠ F, B ≠ D, B ≠ C, F ≠ D, F ≠ E, C ≠ E, C ≠ D, E ≠ D.



9

Exemple: Problème de coloriage de graphe

- Formellement, **un problème de satisfaction de contraintes** (ou CSP pour Constraint Satisfaction Problem) est défini par:
 - Un ensemble fini de **variables** X₁ , ..., X_n .
 - Chaque **variable** X_i a **un domaine Di** de valeurs possibles/permises.
 - Un ensemble fini de **contraintes** C₁ , ..., C_m sur les variables.
 - Une contrainte restreint les valeurs pour un sous-ensemble de variables.

10

CSP: Définition

- Un état (nœud) d'un problème CSP est défini par une **assignation** de valeurs à certaines variables ou à toutes les variables.
 - $\{X_1=v_1, X_2=v_2, \dots\}$
- Une assignation qui ne viole aucune contrainte est dite **consistante** ou **légale**.
- Une assignation **est complète** si elle concerne toutes les variables.
- **Une solution à un problème CSP est une assignation complète et consistante.**
- Parfois, la solution doit en plus **maximiser une fonction objective donnée**.

11

Exemple 2: Colorier une carte

- On vous donne une carte de l'Australie :



- Et on vous demande d'utiliser seulement trois couleurs (**rouge**, **vert** et **bleu**) de sorte que deux états frontaliers n'aient jamais les mêmes couleurs.
- On peut facilement trouver une solution à ce problème en le formulant comme un problème CSP et en utilisant des algorithmes généraux pour CSP.

12

Exemple 2: Colorier une carte

- Formulation du problème CSP :



- Les variables sont les états : $V = \{ WA, NT, Q, NSW, V, SA, T \}$
- Le domaine de chaque variable est l'ensemble des 3 couleurs : **R G B**

WA	NT	Q	NSW	V	SA	T
R	G	B	R	G	B	R

- Contraintes : *Les régions frontalières doivent avoir des couleurs différentes*

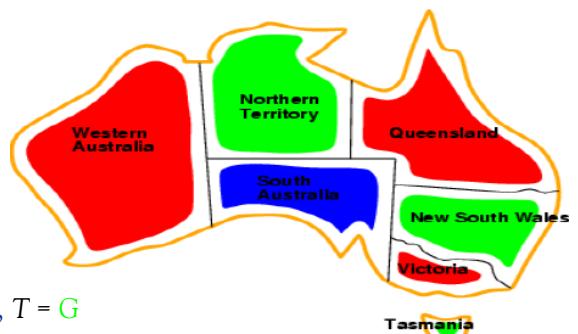
13

✓ $WA \neq NT, \dots, NT \neq Q, \dots$

Exemple 2: Colorier une carte

La **solution** est complète et **consistante**

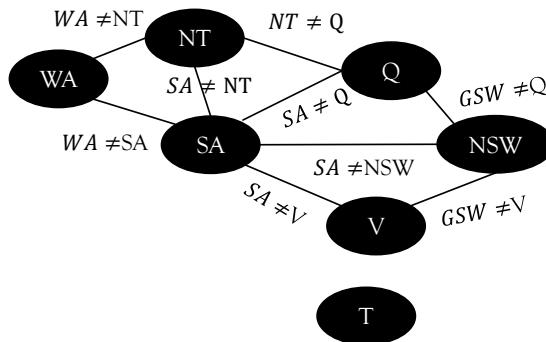
$$\{ WA = R, NT = G, Q = R, NSW = G, V = R, SA = B, T = G \}$$



14

Graphe de contraintes

- Pour des problèmes avec des contraintes binaires (c-à-d., entre deux variables), on peut visualiser le problème CSP par un graphe de contraintes.
- Dans **un graphe de contraintes**: les nœuds sont les variables, et les arcs sont les contraintes



Intérêt des CSP

- La **fonction successeur** et le **test de l'état final** sont **génériques**
 - Les heuristiques sont génériques
 - Le graphe des contraintes peut simplifier le processus de recherche.

Formulation d'un CSP

- **Formulation incrémentale**
 - ✓ **Etat initial:** aucune variable affectée
 - ✓ **Fonction successeur:** affecter une valeur à n'importe quelle variable non encore affectée, à condition que celle là ne génère pas de conflit.
 - ✓ **Test de l'état final:** affectation courante complète
 - ✓ **Coût du chemin:** un coût constant à toutes les étapes.
- **Formulation par états complets:** le chemin menant à la solution n'a pas d'importance.

17

Exemple de problème de 4 reines

- Placer 4 reines sur un échiquier de 4 lignes et 4 colonnes de façon à ce qu'aucune reine ne soit attaquée
- 2 reines sont attaquées si elles se trouvent sur une même diagonale, une même ligne ou une même colonne de l'échiquier



18

CSP (4 reines)

Variables ? (les inconnues du problème)

Domaine ? (les valeurs possibles pour les variables)

Contraintes ?



19

CSP (4 reines): Modélisation

Variables: associer chaque reine i à deux variables L_i (ligne) et C_i (Colonne)

Domaine: $\{1,2,3,4\}$

→ $D(L_1) = D(L_2) = D(L_3) = D(L_4) = D(C_1) = D(C_2) = D(C_3) = D(C_4) = \{1,2,3,4\}$



20

CSP (4 reines): Modélisation

Contraintes :

- Les reines doivent être sur des lignes différentes.
✓ $\text{Clig} = \{L_1 \neq L_2, L_1 \neq L_3, L_1 \neq L_4, L_2 \neq L_3, L_2 \neq L_4, L_3 \neq L_4\}$
- Les reines doivent être sur des colonnes différentes.
✓ $\text{Ccol} = \{C_1 \neq C_2, C_1 \neq C_3, C_1 \neq C_4, C_2 \neq C_3, C_2 \neq C_4, C_3 \neq C_4\}$
- Les reines doivent être sur des diagonales montantes différentes.
✓ $\text{Cdm} = \{C_1 + L_1 \neq C_2 + L_2, C_1 + L_1 \neq C_3 + L_3, C_1 + L_1 \neq C_4 + L_4, C_2 + L_2 \neq C_3 + L_3, C_2 + L_2 \neq C_4 + L_4, C_3 + L_3 \neq C_4 + L_4\}$
- Les reines doivent être sur des diagonales descendantes différentes.
✓ $\text{Cdd} = \{C_1 - L_1 \neq C_2 - L_2, C_1 - L_1 \neq C_3 - L_3, C_1 - L_1 \neq C_4 - L_4, C_2 - L_2 \neq C_3 - L_3, C_2 - L_2 \neq C_4 - L_4, C_3 - L_3 \neq C_4 - L_4\}$
- L'ensemble de contraintes est défini par l'union de ces 4 ensembles $\text{C} = \text{Clig} \cup \text{Ccol} \cup \text{Cdm} \cup \text{Cdd}$



Type de CSP

- CSP avec des domaines finis (et discrets).
 - CSP Booléens: les variables sont vraies ou fausses.
 - CSP avec des domaines continus (et infinis)
 - ✓ Par exemple, problèmes d'ordonnancement avec des contraintes sur les durées.
 - CSP avec des contraintes linéaires.
 - CSP avec des contraintes non linéaires.
 - ...
- ==> Les problèmes CSP sont étudiés de manière approfondies en recherche opérationnelle.

Type de contraintes

La contrainte unaire qui restreint la valeur d'une seule variable,

- ✓ Ex: SA ≠ green

La contrainte binaire qui porte sur deux variables

- ✓ Ex: SA ≠ WA

Les contraintes d'ordre supérieur impliquent au minimum 3 variables

- ✓ Ex: les énigmes de cryptarithmétique, les 4-reines, etc.

23

CSP (4 reines) type de contraintes

- **Contraintes binaires**

- ✓ Les reines doivent être sur des lignes différentes.
 $\text{Clig} = \{L1 \neq L2, L1 \neq L3, L1 \neq L4, L2 \neq L3, L2 \neq L4, L3 \neq L4\}$

- ✓ Les reines doivent être sur des colonnes différentes.
 $\text{Ccol} = \{C1 \neq C2, C1 \neq C3, C1 \neq C4, C2 \neq C3, C2 \neq C4, C3 \neq C4\}$

- **Contraintes quaternaires**

- ✓ **Contraintes de diagonale montante (Cdm)** = $\{C1+L1 \neq C2+L2, C1+L1 \neq C3+L3, C1+L1 \neq C4+L4, C2+L2 \neq C3+L3, C2+L2 \neq C4+L4, C3+L3 \neq C4+L4\}$

- ✓ **Contraintes de diagonale descendante (Cdd)** = $\{C1-L1 \neq C2-L2, C1-L1 \neq C3-L3, C1-L1 \neq C4-L4, C2-L2 \neq C3-L3, C2-L2 \neq C4-L4, C3-L3 \neq C4-L4\}$

24

Plan Chapitre 3

- Rappel
- Description des CSP's
- Exploration avec backtracking**
- Améliorations du backtracking
 - Most constrained variable
 - Most constraining variable
 - Least constraining value
 - Forward checking
 - Propagation des contraintes
 - Cohérence de arcs
- CSP local: Formulation avec min-conflicts.

25

Algorithme Depth-First-Search Naïve pour CSP : Exploration avec Backtracking

- On pourrait être tenté d'utiliser la recherche dans un graphe ([Algorithme en profondeur d'abord](#)) ou un *depth-first-search naïf* avec les paramètres suivants:
 - ✓ Un état est une assignation.
 - ✓ État initial : assignation vide { }
 - ✓ Fonction successeur : assigne une valeur à une variable non encore assignée, en respectant les contraintes.
 - ✓ But : Assignation complète et consistante.
 - En théorie, l'algorithme est général et s'applique à tous les problèmes CSP discrets. La solution doit être complète apparaît à une profondeur n , si nous avons n variables.
 - Cependant, le chemin à la solution est sans importance.
- On peut travailler avec des états qui sont des assignations complètes (consistantes ou non).
- On peut utiliser une méthode de recherche locale (*hill-climbing*, etc.)

26

Limitations de l'approche précédente

- Taille de l'arbre de recherche:
 - Le nombre de branches au premier niveau, dans l'arbre est de n^d , parce que nous avons n variables, chacune pouvant prendre d valeurs.
 - Au prochain niveau, on a $(n-1)d$ successeurs pour chaque noeud.
 - Ainsi de suite jusqu'au niveau n .
 - Le nombre total de branches dans l'arbre de recherche est le produit du nombre de branches à chaque niveau: $n \times d \times (n-1) \times d \times (n-2) \times d \times \dots \times 1 \times d$
 - Cela donne $n! \times d^n$ noeuds générés, pour seulement d^n assignations complètes.
- **Problème:** L'algorithme ignore la commutativité des transitions. Par exemple:
 - SA=R suivi de WA=B est équivalent à WA=B suivi de SA=R.
 - En tenant compte de la commutativité, le nombre de noeuds générés se réduit à d^{n-2}

Limitations de l'approche précédente

Backtracking Search (DFS amélioré)

- Il s'agit d'une **recherche en profondeur** où **chaque noeud représente l'assignation d'une seule variable**.
- Il explore en profondeur jusqu'à une assignation complète, et revient en arrière si une contrainte est violée.
- C'est **l'algorithme de base pour résoudre les CSP**, car il est plus efficace que DFS en évitant de générer toutes les permutations possibles des variables.

- **DFS** explore toutes les possibilités, mais génère trop de noeuds inutiles.
- **DFS** avec backtracking est plus efficace car il évite les chemins non viables et n'exploré que les assignations partielles pertinentes.

Illustration de backtracking-search

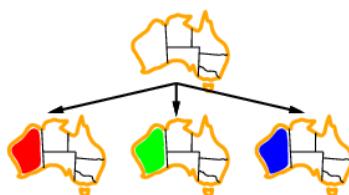
- Étant donnée 3 variables A, B et C définie sur les domaines $D(A)=\{1,2,3\} = D(B)=\{1,2,3\}=D(C) = \{1,2,3\}$ avec les contraintes $A>B$ et $B\neq C$ et $A \neq C$



29

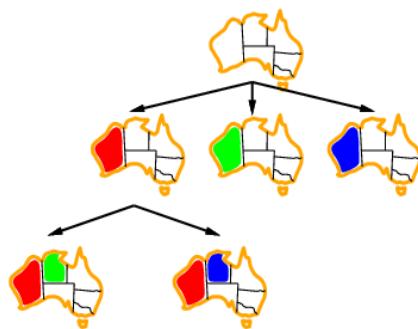
Illustration de backtracking-search

WA	NT	Q	NSW	V	SA	T
■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■



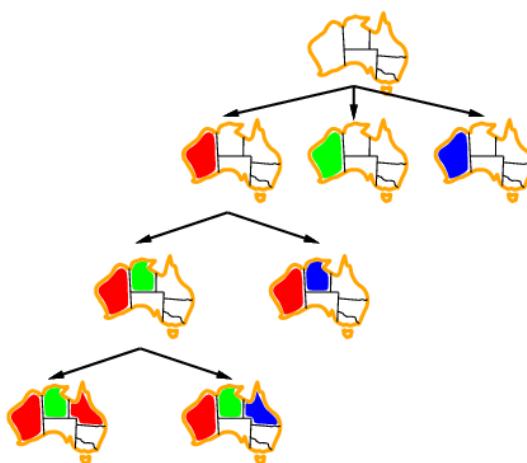
30

Illustration de backtracking-search



31

Illustration de backtracking-search



32

Illustration de backtracking-search

```

function BACKTRACKING-SEARCH(csp) return a solution or failure
    return BACKTRACK({}, csp)
function BACKTRACK(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(var, assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with failure, then
            add {var=value} to assignment
            inferences  $\leftarrow$  INFERENCE(csp, var, value) // e.g., AC-3
            if inferences  $\neq$  failure, then
                add inferences to assignment
                result  $\leftarrow$  BACKTRACK (assignment, csp)
                if result  $\neq$  failure, then return result
            remove {var=value} and inferences from assignment
    return failure

```

L'inférence est une technique utilisée pour réduire l'espace de recherche et éviter d'explorer inutilement des solutions impossibles. Elle consiste à propager les contraintes après avoir fait une assignation, afin d'éliminer des valeurs impossibles des domaines des variables non encore assignées.

33

Inférence

Pourquoi utiliser l'inférence ?

- Lorsqu'on assigne une valeur à une variable, cela peut restreindre les choix pour les autres variables encore non assignées.
- L'idée est d'appliquer des techniques d'inférence pour détecter plus tôt les chemins impossibles, et ainsi éviter d'explorer des branches inutiles dans l'arbre de recherche.

Exemple d'inférence : Arc Consistency (AC-3)

L'algorithme AC-3 (Arc Consistency 3) est l'une des techniques d'inférence les plus utilisées. Il vérifie si une assignation entraîne des réductions dans les domaines des autres variables et supprime les valeurs incohérentes.

Exemple : Coloration de la carte

1. Situation initiale :

1. Région A : {R, V, B}, Région B : {R, V, B} et Région C : {R, V, B}

2. Assignation :

1. On assigne Rouge (R) à la région A. Sans inférence, on continuerait à explorer naïvement toutes les combinaisons possibles pour B et C.

3. Inférence avec AC-3 :

1. Puisque A est Rouge, les régions adjacentes à A ne peuvent plus être Rouge.
2. On met à jour leurs domaines :
 1. Région B : {V, B} (on enlève R)
 2. Région C : {V, B} (on enlève R)
3. Cette réduction du domaine peut éviter de tester inutilement certaines valeurs plus tard. Si cette inférence mène à une contradiction (domaine d'une variable est vide), on revient en arrière immédiatement.

4. Propagation des contraintes :

1. Si on assigne ensuite Vert (V) à B, alors C ne pourra plus prendre V, il lui restera donc Bleu (B) comme seule option.

34

Backtracking (BT) Intelligent

- **backtracking classique (BT)** visite encore et encore les mêmes régions de l'arbre de recherche, car il a une vision très locale du problème



=> Si l'algorithme revient en arrière à chaque niveau et explore chaque possibilité, il peut revisiter les mêmes zones de l'arbre plusieurs fois.

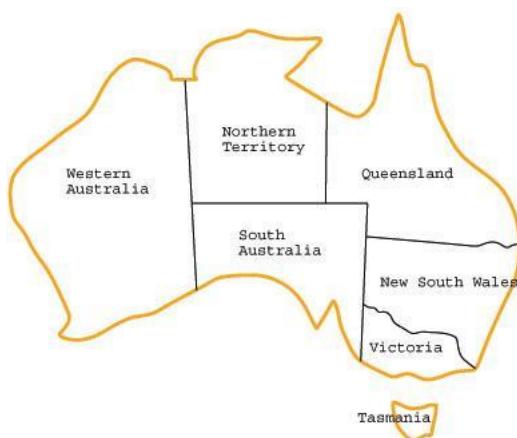
Solution: Le **backtracking intelligent** vise à améliorer l'efficacité du BT en réduisant le nombre de répétitions et de visites inutiles dans les mêmes zones de l'arbre de recherche.

- Une façon de se débarrasser du problème utilise le **backtracking intelligent** algorithms
 - **Les techniques de Backjumping (BJ) et ses variantes:** Backjumping (BJ), Conflict BJ, DB, Graph-based BJ, Learning:
- **Backjumping (BJ)** est différent du BT dans ce qui suit:
 - Quand BJ atteint un blocage il ne fait pas un backtrack à la dernière variable immédiate. Il fait un backtrack à la variable la plus profonde dans l'arbre de recherche qui est en conflit avec la variable courante.

BJ vs. BT

- Revenons au problème de coloriage

Avec les trois couleurs : r, g, b.



BJ vs. BT

- Considérons ce que fait le BT dans le problème de coloriage

- Assumons que les variables sont assignées dans l'ordre : Q, NSW, V, T, SA, WA,
- Assumons que nous avons atteint l'affectation partielle :

$$Q = \text{red}, \text{NSW} = \text{green}, V = \text{blue}, T = \text{red}$$



- En essayant d'affecter une valeur à SA, nous découvrons que toutes les valeurs du domaine ne sont plus permises.
 - **Bolcage!**
- BT va effectuer un backtrack pour essayer une nouvelle valeur de la variable T
 - Pas une bonne idée!

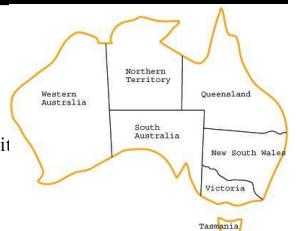
37

BJ vs. BT

- BJ a une approche plus intelligente pour le backtracking

- Il nous dit de revenir à une des variables responsables du blocage.
- L'ensemble de ces variables est appelé **conflict-set** (ensemble de variables responsables du conflit)
 - Le conflict-set de SA est {Q, NSW, V}
 - BJ effectue un saut en arrière vers la variable la plus profonde dans le conflict-set de la variable où le blocage est arrivé.
 - La plus profonde = la plus récemment visitée.

==> En effectuant ce "saut", BJ économise du temps en évitant de revenir en arrière à des niveaux déjà explorés et qui ne sont pas responsables du conflit.



- CBJ, DB, Graph-based BJ, Learning, Backmarking

38

Plan Chapitre 3

- Rappel
- Description des CSP's
- Exploration avec backtracking
- Améliorations du backtracking
 - Most constrained variable
 - Most constraining variable
 - Least constraining value
 - Forward checking
 - Propagation des contraintes
 - Cohérence de arcs
- CSP local: Formulation avec min-conflicts.

39

Amélioration de backtracking-search

- Sans heuristiques, l'algorithme est limité.
- Des heuristiques générales peuvent améliorer l'algorithme significativement :
 - Choisir judicieusement la prochaine variable:
SELECT-UNASSIGNED-VARIABLE
 - Choisir judicieusement la prochaine valeur à assigner:
ORDER-DOMAIN-VALUES
 - Faire des inférences pour détecter plus tôt les assignations conflictuelles
INFERENCE

choisir la prochaine variable à assigner en fonction de certains critères qui augmentent les chances de succès: exemple de critère Minimum Remaining Values (MRV)

Par exemple choisir la valeur qui limite le moins les choix pour les autres variables. Cette heuristique minimise les conflits futurs en réduisant la taille des domaines des autres variables.

propager les conséquences d'une assignation pour identifier rapidement les assignations conflictuelles et éviter des explorations inutiles.

40

Amélioration de backtracking-search

- L'amélioration des performances des algorithmes de **Backtracking Search** repose sur l'utilisation de **heuristiques intelligentes** pour choisir les variables et les valeurs, ainsi que sur des techniques de **pruning** pour éviter des explorations inutiles:
 - Quelle variable doit être affectée à la prochaine étape?
 - Dans quel ordre ses valeurs devraient être choisies?
 - Peut-on détecter les inévitables échecs plutôt?
- **Heuristiques:**
 1. Most constrained variable
 2. Most constraining variable
 3. Least constraining value
 4. Forward checking

41

Heuristique 1: Most constrained Variable

Quelle variable a le moins d'options possibles ?

- À chaque étape, choisir la variable avec le moins de valeurs consistantes restantes.
 - C.-à-d., la variable « posant le plus de restrictions ».
 - Appelé aussi **Minimum Remaining Value (MRV)**.

Objectif: réduire rapidement l'espace de recherche en traitant les cas les plus restrictifs en premier.

Exemple :

- Variables : A, B, C
- Domaines :
 - A : {1, 2, 3}
 - B : {2} (domaine très restreint → MRV la choisira en premier)
 - C : {1, 2, 3, 4}

On assigne d'abord B car son domaine est le plus petit (1 valeur).

42

Heuristique 2: Most constraining Variable

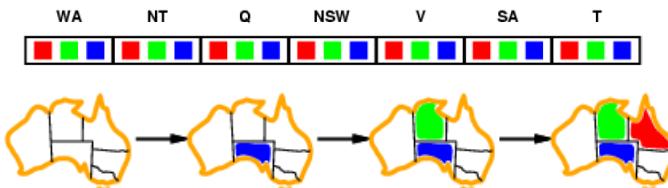
Quelle variable a le plus d'impact sur les autres ?

- À chaque étape, choisir la variable la plus contraignante.
 - C-à-d., la variable « avec le plus grand nombre de contraintes ».
 - Appelée aussi « *Degree heuristic* ».

Objectif : Choisir la variable qui participe au plus grand nombre de contraintes avec d'autres variables

Critère : Sélectionner la variable ayant le degré le plus élevé (nombre de contraintes binaires ou arité supérieure impliquant cette variable).

- Illustration:



Heuristique 3: Least constraining value

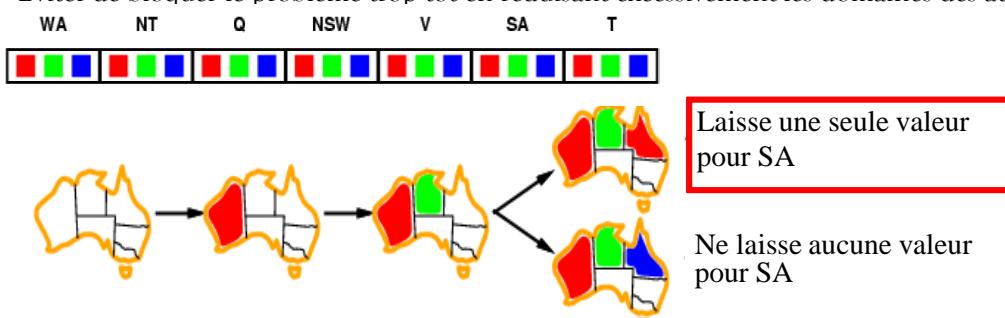
Si j'assigne cette valeur à la variable, combien de choix restent possibles pour les autres variables ?

- L'ordre des valeurs affectées peut aussi améliorer les performances.
- Tu testes plusieurs valeurs et sélectionnes celle qui laisse **le plus de possibilités aux autres variables**. C'est une approche utile pour éviter les impasses dans la recherche de solutions..

Objectif:

- Garder un maximum d'options ouvertes pour les variables restantes.

- Éviter de bloquer le problème trop tôt en réduisant excessivement les domaines des autres variables



Heuristique 4: Forward Checking

Déetecter le plus tôt possible les assignations conflictuelles.

- Le **Forward Checking** est une technique qui **prédit les échecs** dès qu'on affecte une variable, en réduisant immédiatement le domaine des variables non encore assignée
- L'idée de **forward-checking** (*vérification avant*) est :
 - P1:** Chaque fois qu'une variable est assignée, vérifier la cohérence de toutes les variables en contrainte avec elle.
 - P2:** Attention: si la valeur est assignée à X, on vérifie juste la cohérence des variables Y telle que il y a une contrainte impliquant X et Y. Par contre, si ce faisant, Y est modifié, on ne vérifiera pas l'impact de cette modification sur les contraintes impliquant Y!

Principe

Quand on affecte une valeur à une variable :

- On **met à jour les domaines** des variables non encore assignées en éliminant les valeurs **incompatibles**.
- Si une variable non assignée n'a **plus aucune valeur possible**, alors on détecte un ⁴⁵échec immédiatement et on revient en arrière (**backtracking**).

Heuristique 4: Forward Checking

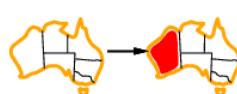
Exemple



Domaines initiaux

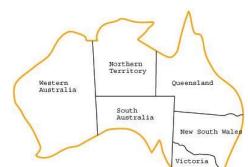
WA	NT	Q	NSW	V	SA	T
Red	Green	Blue	Green	Blue	Green	Blue

- Supposons que l'on choisisse au départ la variable WA (première étape de la récursivité de *backtracking-search*). Considérons l'assignation WA=Rouge. On voit ici le résultat de *forward-checking*.



Domaines initiaux
Après WA=Red

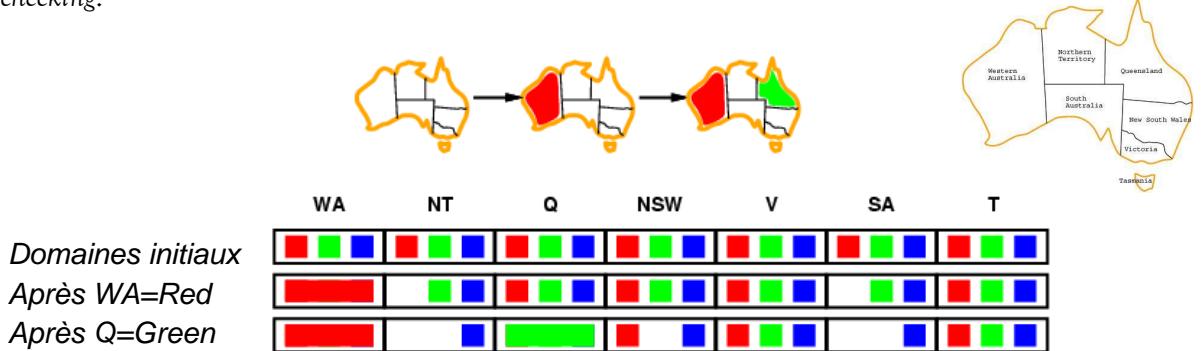
WA	NT	Q	NSW	V	SA	T
Red	Green	Blue	Green	Blue	Green	Blue



46

Heuristique 4: Forward Checking

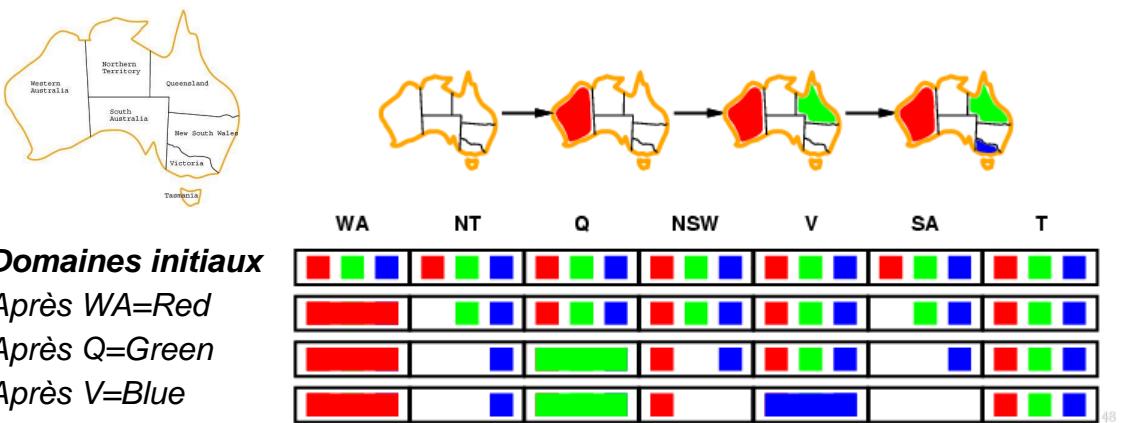
- Supposons maintenant que l'on choisisse la variable Q à la prochaine étape de la récursivité de *backtracking-search*. Considérons l'assignation Q=Vert. On voit ici le résultat de *forward-checking*.



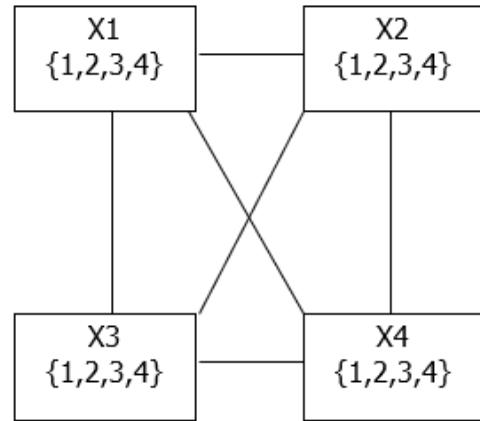
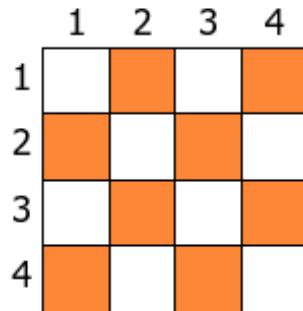
47

Heuristique 4: Forward Checking

- Supposons maintenant que l'on choisisse la variable V à la prochaine étape de la récursivité de *backtracking-search*. Considérons l'assignation V=Bleu. On voit ici le résultat de *forward-checking*.

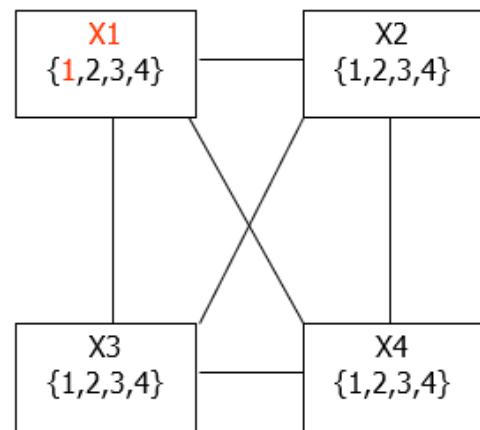
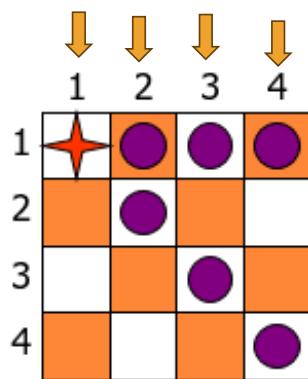


Exemple : problème des 4-reines avec FC



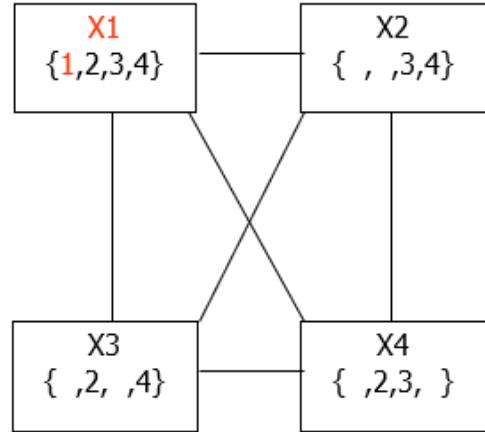
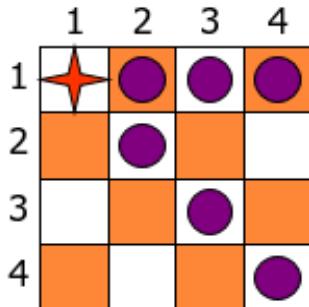
49

Exemple : problème des 4-reines avec FC



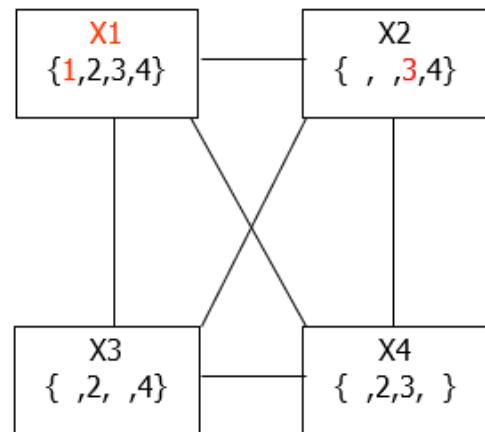
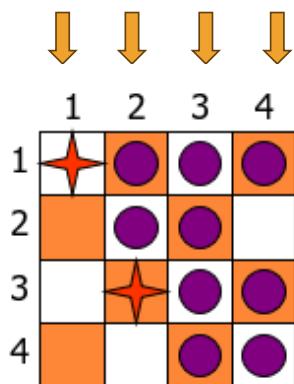
50

Exemple : problème des 4-reines avec FC



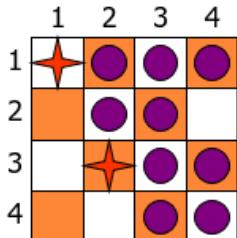
51

Exemple : problème des 4-reines avec FC

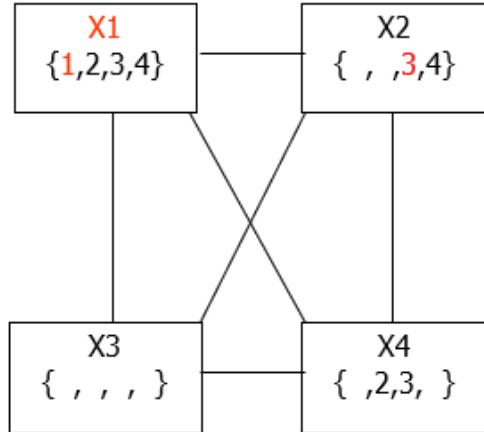


52

Exemple : problème des 4-reines avec FC

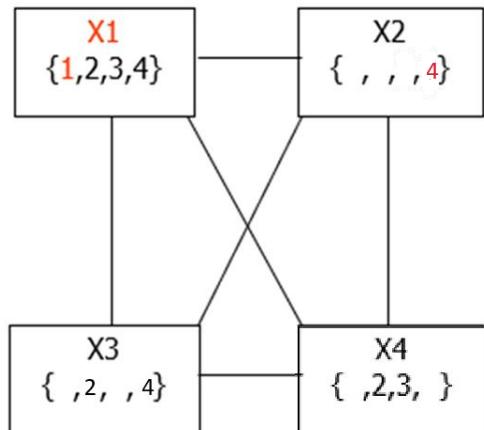
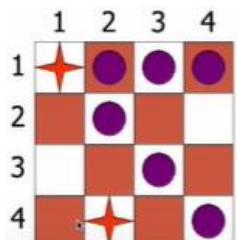


Backtrack



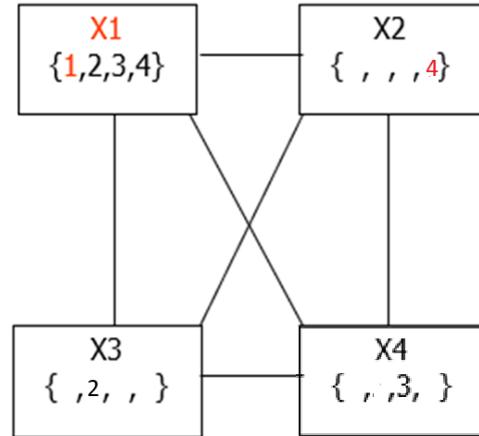
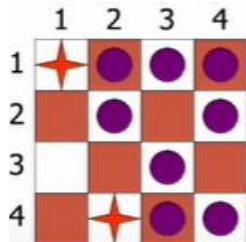
53

Exemple : problème des 4-reines avec FC



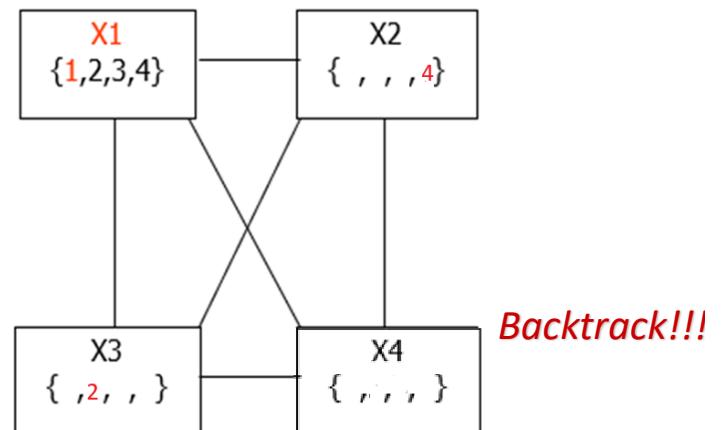
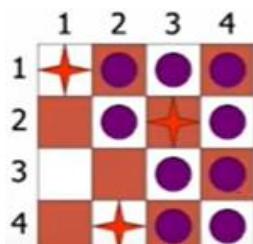
54

Exemple : problème des 4-reines avec FC



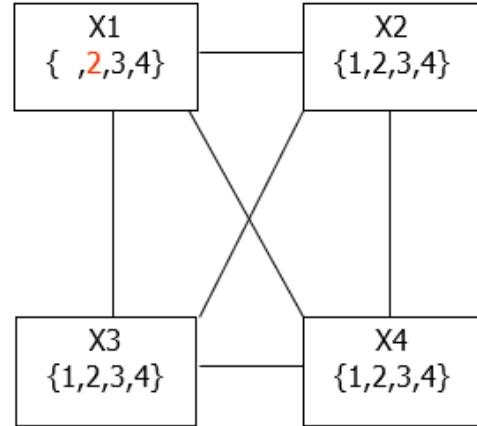
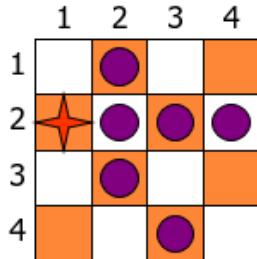
55

Exemple : problème des 4-reines avec FC



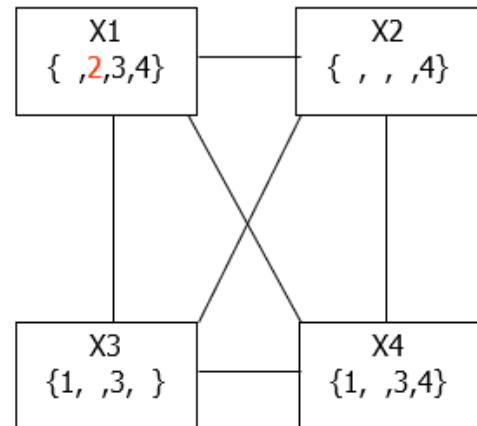
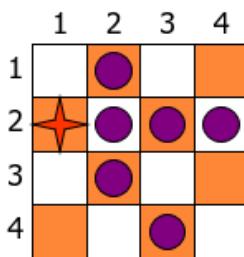
56

Exemple : problème des 4-reines avec FC



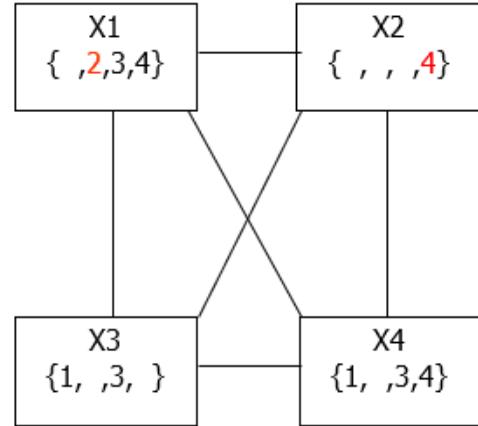
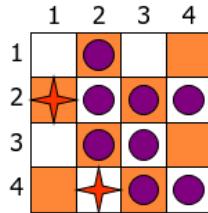
57

Exemple : problème des 4-reines avec FC



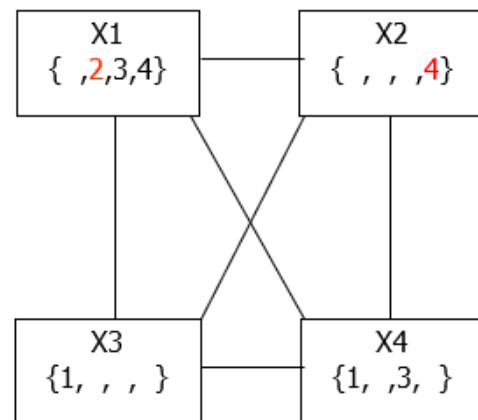
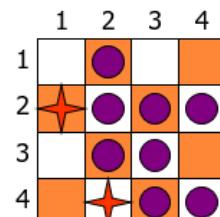
58

Exemple : problème des 4-reines avec FC



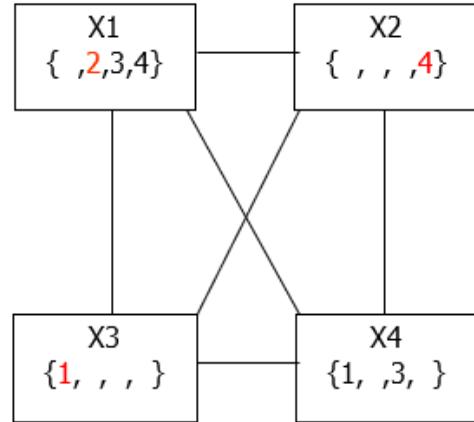
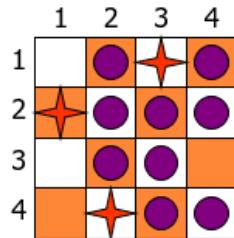
59

Exemple : problème des 4-reines avec FC



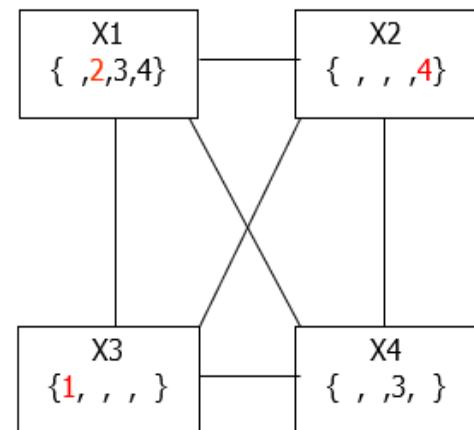
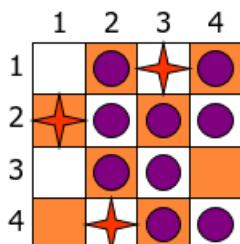
60

Exemple : problème des 4-reines avec FC



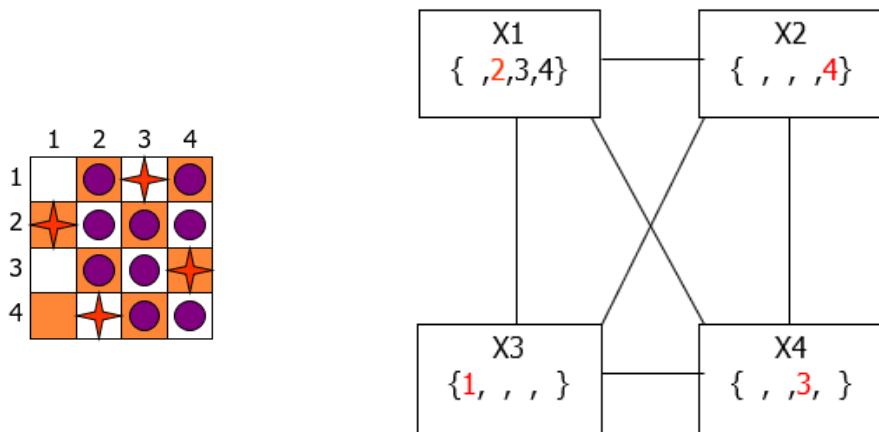
61

Exemple : problème des 4-reines avec FC



62

Exemple : problème des 4-reines avec FC



63

En résumé

	Most constrained variable	Most constraining Variable	Least Constraining Value (LCV)	Forward Cheking
Principe	Choisir la variable qui a le moins de valeurs possibles restantes (variable la plus contrainte).	Choisir la variable qui contraint le plus les autres.	Lors de l'affectation d'une valeur à une variable, choisir la valeur qui laisse le plus d'options disponibles pour les autres variables.	Dès qu'une valeur est assignée à une variable, vérifier les variables voisines pour s'assurer qu'elles ne deviennent pas non satisfaites.
But	Réduire rapidement l'espace de recherche en concentrant les efforts sur les variables les plus restreintes.	Réduire l'impact des contraintes sur d'autres variables en traitant en premier celles qui contraignent le plus.	Minimiser les effets négatifs des choix de valeur sur les autres variables.	Détecter tôt les échecs potentiels pour éviter d'explorer des branches inutiles.

64

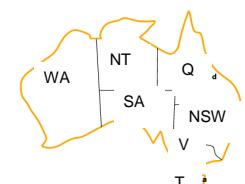
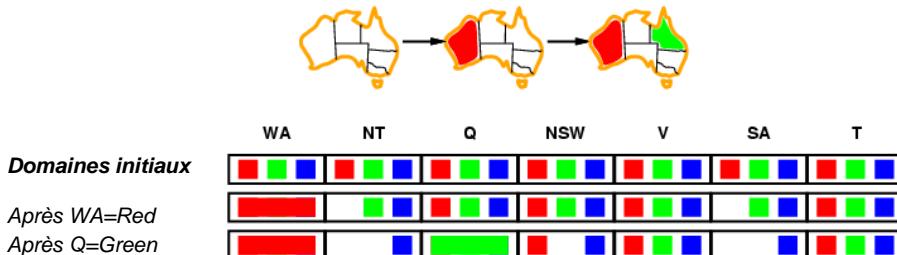
Propagation de contraintes

- Forward checking propage l'information des variables assignées vers les variables non assignées, **mais ne détecte pas les conflits locaux entre variables non assignées**



- Le **Forward Checking (FC)** est une technique efficace pour maintenir **la consistance locale** dans les problèmes de satisfaction de contraintes (CSP), mais il présente certaines **limites**:
 - ✓ Le Forward Checking ne maintient la consistance qu'entre la **variable courante** et les **variables futures** (non encore assignées).
 - ✓ Il ne vérifie pas les contraintes entre les **variables futures elles-mêmes**. Cela signifie qu'il peut manquer des conflits qui surviennent entre ces variables.
- Ces limites sont souvent surmontées par des techniques plus avancées, comme la **propagation des contraintes** (par exemple, AC-3 ou **Maintenance de la Consistance d'Arc**).

Propagation de contraintes



- Revenons à l'étape de backtracking-search, après que nous ayons choisi la variable Q et nous voudrons assigné la valeur bleu
 - ✓ On voit ici le résultat de forward-checking
 - ✓ Forward-checking ne propage pas la modification du domaine NT vers SA pour constater que NT et SA ne peuvent pas être en bleu ensemble!
- Limite de Forward Checking :** Forward Checking ne propagera pas l'impact de cette assignation sur NT et SA. Cela signifie que si la contrainte indique qu'elles ne peuvent pas être colorées en bleu en même temps que Q, cette incohérence ne sera pas détectée.

Propagation de contraintes

Propagation des contraintes :

Ce qui manque avec **Forward Checking**, c'est la **propagation des contraintes**. La propagation des contraintes permettrait de détecter que si **Q** est colorée en bleu, alors **NT** et **SA** ne peuvent pas être colorées en bleu en même temps, et il est donc nécessaire de modifier leurs domaines respectifs (réduire les options disponibles pour **NT** et **SA**).

- ➔ Met à jour tous les domaines des variables en fonction des contraintes, détectant ainsi des conflits
- ➔ La propagation des contraintes permet de vérifier les contraintes localement.

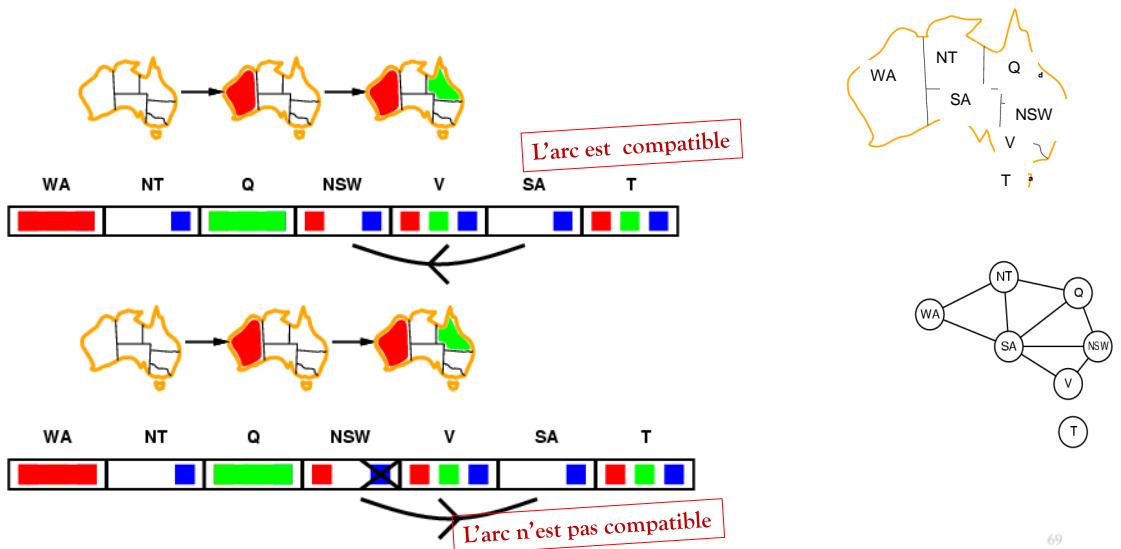
67

Cohérence des arcs (Arc consistency)

- **Arc consistency** est la forme de propagation de contraintes la plus simple
 - ✓ Vérifie la consistance entre les arcs.
 - ✓ C.-à-d., la consistance des contraintes entre deux variables.
- L'arc $X \rightarrow Y$ est consistant si et seulement si
 - ✓ Pour chaque valeur x de X il existe au moins une valeur permise de y .
 - ✓ Inversement, pour chaque valeur y de Y , il existe au moins une valeur x de X telle que la contrainte est satisfaite.
- Si une valeur x dans X n'a aucune valeur correspondante y dans Y , alors x est supprimée du domaine de X

68

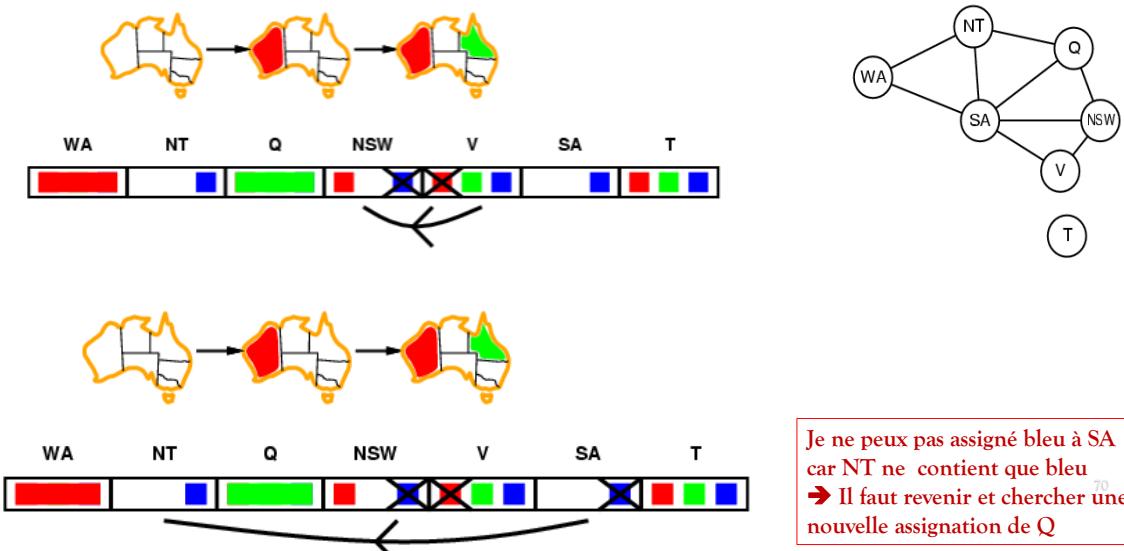
Cohérence des arcs (Arc consistency)



69

Solution: Si une variable perd une valeur, ses voisins doivent être revérifiés.

Cohérence des arcs (Arc consistency)



Cohérence des arcs (Arc consistency)

- L'arc-consistance ne détecte pas toutes les inconsistances possibles
- Pour aller plus loin dans la propagation des contraintes, on peut tester la cohérence à un niveau supérieur, ce qui nous amène à la notion de **k-consistance (AC-k)**.
- Plus les vérifications de consistance sont d'ordre élevé, plus on a une :
 - Augmentation de la complexité en temps
 - Diminution du facteur de branchement

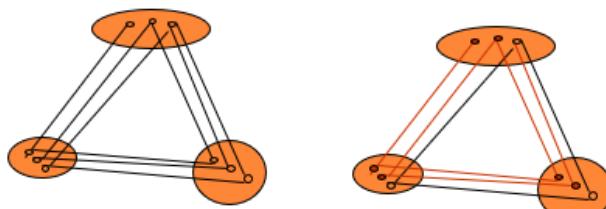
71

La consistance d'arc (AC) : AC1

L'algorithme **AC-1** (Arc Consistency 1) est l'une des premières méthodes de **filtrage par consistance d'arc** (Arc Consistency, AC) utilisée

suppression des valeurs qui ne vérifient pas la propriété (AC)

=> Le **filtrage par consistance d'arc** consiste à éliminer de chaque domaine de variables les valeurs qui ne respectent pas la contrainte entre cette variable et ses voisins dans le réseau de contraintes.



72

La consistance d'arc (AC) : AC1

- L'AC-1 est un algorithme simple pour établir la consistance d'arc. Il fonctionne en **répétant des passes** sur toutes les contraintes jusqu'à ce qu'aucune valeur ne soit supprimée des

procédure AC-1(G)

répéter

```

        changement ← faux
        pour chaque arc  $(i,j) \in G$  faire
            changement ← réviser( $(i,j)$ ) ou changement
        fin pour
    jusqu'à non (changement)
fin AC-1
    
```

procédure Réviser((i,j))

```

        changement ← faux
        pour  $d_i \in D_i$  faire
            si  $\exists d_j \in D_j$  telle que  $(d_i, d_j) \in R_k$ 
            alors supprimer  $d_i$  de  $D_i$ 
            changement ← vrai
        fin si
    fin pour
    retourner changement
fin
    
```

La fonction **Réviser** vérifie chaque contrainte dans un arc donné et s'assure que pour chaque valeur dans le domaine de la variable concernée, il existe au moins une valeur dans le domaine de l'autre variable qui satisfait la contrainte.

73

L'algorithme AC1

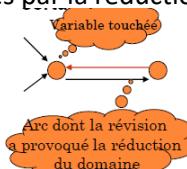
Quels sont les défauts de AC-1?

- L'AC-1 **révise tous les arcs** à chaque passe, même si aucun domaine n'a été réduit lors de la passe précédente.
- Cela entraîne une **surcharge computationnelle** inutile, car de nombreuses révisions ne contribuent pas à la réduction des domaines

Quels sont les arcs à reconSIDÉRER ?

Manque de ciblage :

- Lorsqu'un domaine est réduit, l'AC-1 **révise tous les arcs incidents** (entrants et sortants) à la variable touchée, même si certains arcs ne sont pas affectés par la réduction.
- Ignorer les arcs incidents de la variable** : Les **arcs incidents** (c'est-à-dire les arcs sortants de la variable touchée) doivent être **ignorés**. Ils n'ont pas d'impact direct sur la consistance des autres variables, car ils ne sont pas affectés par la réduction du domaine de la variable en question celle touché. (**arcs sortants**)



74

L'algorithme AC3 [Mackworth 1977]

- utiliser une file pour mémoriser les arcs à (re-)réviser
- enfiler uniquement les arcs affectés par la réduction des domaines.

```

procedure AC-3(G)
    Q ← {(i,j) | (i,j) ∈ arcs(G), i < j}      % file pour les arcs à réviser
    tant que Q non vide faire
        sélectionner et supprimer (k,m) de Q
        si réviser((k,m)) alors
            Q ← Q ∪{(i,k) | (i,k) ∈ arcs(G), i ≠ k, i ≠ m}
        fin si
    fin tant que
fin

```

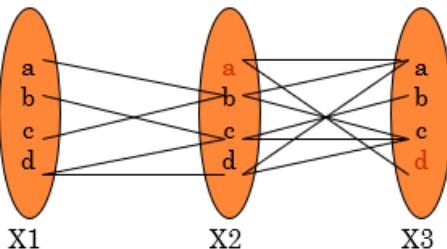
☞ AC3 est l' algorithme le plus utilisé

75

L'algorithme AC3 : observations

Révision d'un arc : de nombreuses paires de valeurs sont testées

Ces tests sont répétés à chaque fois qu'un arc est révisé



- Quand l'arc $\langle X_2, X_1 \rangle$ est révisé, la valeur a est supprimé du domaine de X_2
- le domaine de X_3 doit être exploré pour déterminer si les valeurs a, b, c et d perdent leur support dans X_2



il est nécessaire de vérifier si cela affecte les autres variables liées à X_2 , en particulier X_3 . Cela revient à **propager l'effet de la modification** dans le réseau de contraintes.

Observation

Il n'est pas nécessaire d'examiner les valeurs a, b et c de X_3 (elles admettent un autre support autre que a dans X_2)

76

L' algorithme AC3 : observations

Algorithme AC-3(csp) // retourne le CSP simplifié et un booléen vrai si pas de conflit

1. $file_arcs = \text{ARCS-DU-CSP}(csp)$
2. tant que $file_arcs$ n'est pas vide
 3. $(X_i, X_j) = \text{POP}(file_arc)$ // retire premier arc de la file
 4. $\text{changé}, csp = \text{RÉVISER}(X_i, X_j, csp)$ // vérifie la compatibilité de l'arc
 5. si changé
 6. si $\text{DOMAINE}(X_i, csp)$ est vide, retourner (void, faux)
 7. pour chaque X_k dans $\text{VOISINS}(X_i, csp)$
 8. si $X_k \neq X_j$, ajouter (X_k, X_i) dans $file_arcs$
 8. retourner (csp, vrai)

on suppose que
 csp est passé par copie

Algorithme RÉVISER(X_i, X_j, csp) // réduit le domaine de X_i en fonction de celui de X_j

1. $\text{changé} = \text{faux}$
2. pour chaque x dans $\text{DOMAINE}(X_i, csp)$
 3. si aucun y dans $\text{DOMAINE}(X_j, csp)$ satisfait contrainte entre X_i et X_j
 4. enlever x de $\text{DOMAINE}(X_i, csp)$ // ceci change la variable csp
 5. $\text{changé} = \text{vrai}$
4. retourner ($\text{chanaé}.csp$)

77

Résumé

- Les problèmes CSP sont des problèmes de recherche dans un espace d'assignation de valeurs à des variables.
- *Backtracking-search* = depth-first-search avec une variable assignée par nœud.
- L'ordonnancement des variables et celui de l'assignation des valeurs aux variables jouent un rôle significatif dans la performance.
- *Forward-checking* empêche les assignations qui conduisent à un conflit.
- La propagation des contraintes (par exemple, *arc-consistency*) détecte les inconsistances locales.
- Les méthodes les plus efficaces exploitent la structure du domaine.
- Application surtout à des problèmes impliquant l'ordonnancement des tâches.

78

Plan Chapitre 3

- Rappel
- Description des CSP's
- Exploration avec backtracking
- Améliorations du backtracking
 - Most constrained variable
 - Most constraining variable
 - Least constraining value
 - Forward checking
 - Propagation des contraintes
 - Cohérence de arcs
- CSP local: Formulation avec min-conflicts.

79

Exploration locale pour les CSP's: Min-Conflicts

1. Sélection de la variable

On choisit **au hasard** une variable qui est en conflit (c'est-à-dire qui ne satisfait pas au moins une contrainte).

2. Sélection de la valeur avec Min-Conflicts

Une fois la variable sélectionnée, on lui attribue **la valeur qui réduit au maximum le nombre de conflits** avec les autres variables.

i.e., executer hill-climbing avec $h(n) = \text{nombre total de contraintes violées} = \text{nombre de conflits}$

80

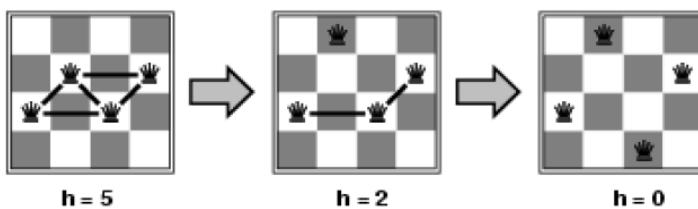
Exemple : 4 reines

Etats: reines dans 4 colonnes ($4^4 = 256$ états)

Actions déplacer une reine dans une colonne

Test du but: pas d'attaque

Evaluation: $h(n) = \text{nombre d'attaques}$



81

CSP et Exploration locale dans le monde réel

- Efficacité sur certains problèmes complexes

- Min-Conflicts est particulièrement efficace pour des problèmes où une solution existe et où les conflits sont localisés.
 - Ex : Planification des observations du télescope Hubble, où il faut ajuster des créneaux sans perturber toute la planification.

- Adaptabilité aux changements dynamiques

- Min-Conflicts permet de modifier rapidement une solution existante en effectuant des ajustements locaux.
 - Ex : Réaménagement des plans de vol en cas de mauvais temps, où il faut minimiser l'impact des retards sur l'ensemble du programme

82