Programmation Par Contraintes : Algorithmes de Résolution

Benjamin NGUYEN
4A-STI INSA Centre Val de Loire

D'après C. Solnon
INSA Lyon, équipe Modèles Multirésolution, Discrets et Combinatoires (M2DisCo)

Survol

Résumé des épisodes précédents

→ Introduction du formalisme des CSP, offrant un cadre structurant pour modéliser des problèmes exprimés en terme de contraintes.

→ Entraînement à la modélisation de tels problèmes

→ Résolution de problèmes en utilisant Choco et Java (Cappuccino?)

Dans ce cours

Présentation et étude d'algorithmes permettant de résoudre, de façon générique de tels problèmes.

Restriction

Les CSPs sur des domaines finis, et pas d'optimisation.

Principe

Une exploration **méthodique** de toutes les affectations possibles jusqu'à soit trouver une solution, soit démontrer qu'il n'existe pas de solution (inconsistance du CSP).

Plan

1. Algorithme « Generate and Test »
2. Analyse critique de « GET » et notion d'espace de recherche d'un CSP
3. Algorithme « Simple Backtrack »
4. Algorithme « Look-Ahead »
5. Intégration d'heuristiques

1- Algorithme « Generate and Test »

« Génère et Teste »

Principe

- **Façon simple et naïve :**
  Enumérer toutes les affectations possibles jusqu'à en trouver une qui satisfasse toutes les contraintes.

- Approche dite « exhaustive »

- Implémentation sous forme de fonction récursive :

```
générEtTeste(A, (X, D, C))
```

- Dans cette fonction, `A` contient une affectation partielle et (`X, D, C`) décrit le CSP à résoudre.
- Au premier appel, A sera vide. La fonction retourne `vrai` si on peut étendre l'affectation partielle en une affectation totale consistante (i.e. une solution) et `faux` sinon.

Algorithme

```
Fonction genereEtTeste(A,(X,D,C)) retourne un booléen

Précondition :   (X,D,C) = un CSP sur les domaines finis,
                 A = une affectation partielle pour (X,D,C)

Postrelation :   Retourne vrai si l'affectation partielle A peut être
                 étendue en une solution pour (X,D,C), faux sinon

début
   si toutes les variables de X sont affectées à une valeur dans A alors
   /* A est une affectation totale */
        si A est consistante alors
        /* A est une solution */
                retourner vrai

        sinon
                retourner faux
        finsi
   sinon /* A est une affectation partielle */
        choisir une variable Xi de X qui n'est pas encore affectée à une
   valeur dans A
        pour toute valeur Vi appartenant à D(Xi) faire
                si genereEtTeste(A U {(Xi,Vi)}, (X,D,C)) = vrai alors
                        retourner vrai
                finsi
        finpour
        retourner faux
   finsi
fin
```

Exemple d'exécution

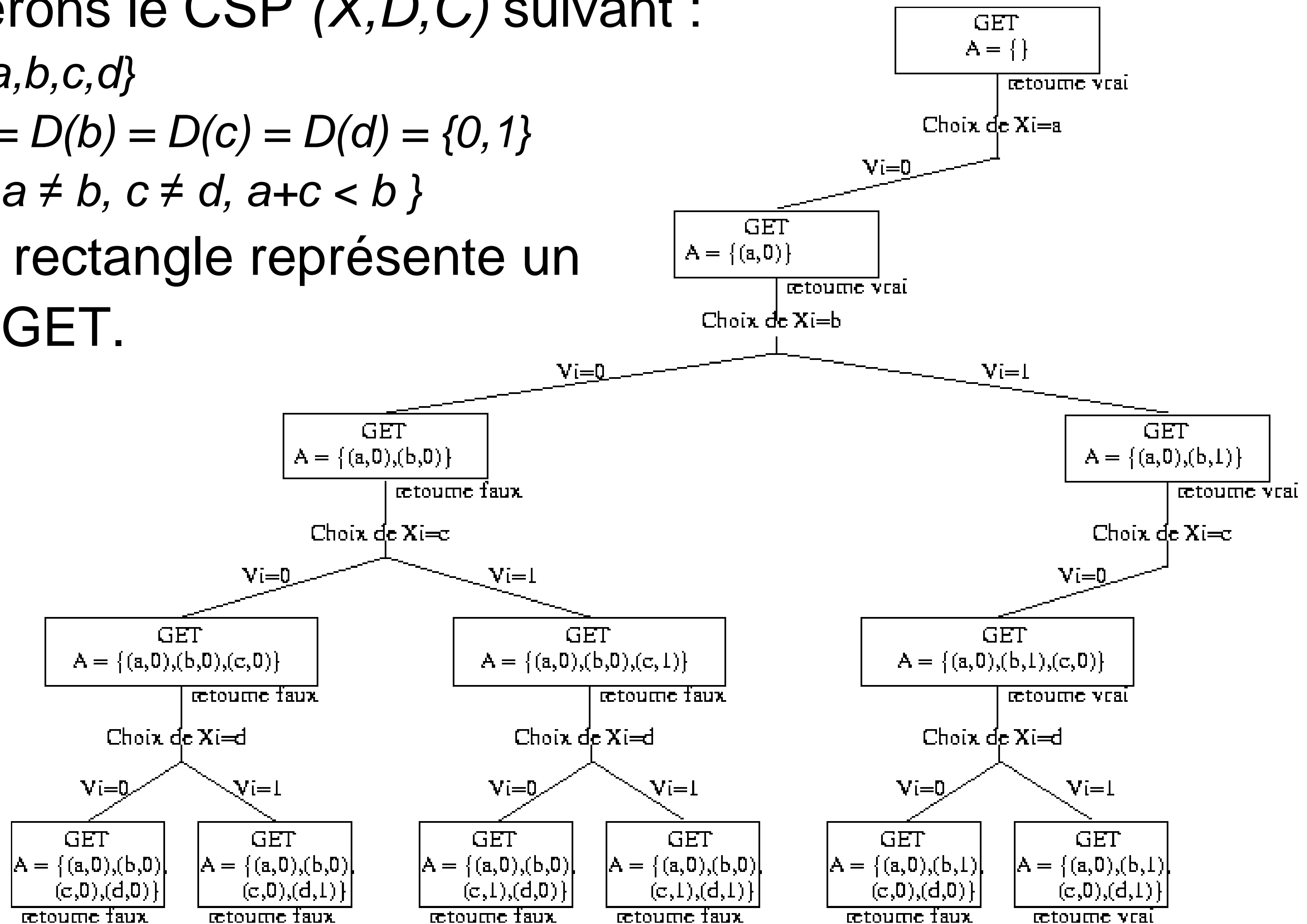
Considérons le CSP *(X,D,C)* suivant :

$X = \{a,b,c,d\}$

$D(a) = D(b) = D(c) = D(d) = \{0,1\}$

$C = \{ a \neq b,\ c \neq d,\ a+c < b \}$

Chaque rectangle représente un appel à GET.



Problèmes ?

Comme on le voit, GET calcule des affectations totales, mais pourrait en fait s'arrêter avant …

2- Analyse critique de « GET » et notion d’espace de recherche d’un CSP

Espace de recherche 1/2

L'algorithme "génère et teste" que nous venons de voir *énumère l'ensemble des affectations complètes possibles, jusqu'à en trouver une qui soit consistante.*

L'ensemble des affectations complètes est appelé l'*espace de recherche du CSP*.

Première limite : Si le domaine de certaines variables contient une infinité de valeurs, alors cet espace de recherche est infini et on ne pourra pas énumérer ses éléments en un temps fini.

Néanmoins, même en se limitant à des domaines comportant un nombre fini de valeurs, l'espace de recherche est souvent de taille tellement importante que l'algorithme "génère et teste" ne pourra se terminer en un temps "raisonnable".

Espace de recherche 2/2

- L'espace de recherche d'un CSP (X,D,C) comportant *n* variables ($X = \{X_1, X_2, ..., X_n\}$) est défini par
  $E = \{ \{(X_1,v_1), (X_2,v_2), ..., (X_n,v_n)\}$ *| quelquesoit i compris entre 1 et n,* v_i *est un élément de* $D(Xi)$ *}*
  et le nombre d'éléments de cet espace de recherche est défini par
  $|E|=|D(X_1)|*|D(X_2)|* ... *|D(X_n)|$ de sorte que, si tous les domaines des variables sont de la même taille *k* (autrement dit, $|D(X_i)|=k$), alors la taille de l'espace de recherche est $|E| = k^n$
- Le nombre d'affectations à générer croit de façon *exponentielle* en fonction du nombre de variables du problème !
- Ordre de grandeur (cf Sudoku) : 80 variables, 10 possibilités, 10^9 affectations par seconde (GHz) = $10^{80}/10^9 = 10^{71}$ secondes >>> l'age de l'univers !
- NB : à 30 variables et 2 possibilités le temps de résolution est de l'ordre de la seconde.

Pistes d’améliorations 1/3

- Si le nombre de variables est > 30 alors on ne peut appliquer cet algorithme « bête ». Il faut **réduire l’espace de recherche**.
- **Idées :**

  ne développer que les affectations partielles consistantes : dès lors qu'une affectation partielle est inconsistante, il est inutile de chercher à l'étendre en une affectation totale puisque celle-ci sera nécessairement inconsistante. Par contre il faut pouvoir détecter une telle affectation !

  → **algo simple backtrack**

Pistes d'améliorations 2/3

Réduire les tailles des domaines des variables en leur enlevant les valeurs "incompatibles" : pendant la génération d'affectations, on filtre le domaine des variables pour ne garder que les valeurs "localement consistantes" avec l'affectation en cours de construction, et dès lors que le domaine d'une variable devient vide, on arrête l'énumération pour cette affectation partielle

→ algo **anticipation**

Pistes d’améliorations 3/3

Introduire des "heuristiques" pour "guider" la recherche : lorsqu'on énumère les affectations possibles, on peut essayer d'énumérer en premier celles qui sont les plus "prometteuses", en espérant ainsi tomber rapidement sur une solution.

Autres idées

- Identifier la cause de l'échec pour retourner là où cette valeur a été instanciée. **Intelligent Backtrack**.
- Exploiter des connaissances sur le types de contraintes utilisées pour réduire l'espace de recherche
- Exemple :

  X = {a, b, c } D = entiers C = {2a + 4b = 2c +3} → n'admet pas de solutions

Homme vs machine

Ce genre de raisonnement demande de l'intelligence, ou pour le moins des connaissances.

De fait, l'homme est capable de résoudre des problèmes très combinatoires en raisonnant (en utilisant son "expérience" et des connaissances plus ou moins explicitées).

Un exemple typique est le jeu d'échec : les grands joueurs d'échecs n'envisagent pour chaque coup à jouer que très peu de combinaisons (les meilleures évidemment !), éliminant par des raisonnements souvent difficiles à expliciter un très grand nombre de combinaisons moins intéressantes.

Cela explique le fait que, malgré leur capacité à envisager un très grand nombre de combinaisons, les ordinateurs ne sont pas forcément plus forts que ces grands joueurs.

3- Algorithme « Simple Backtrack »

Idée

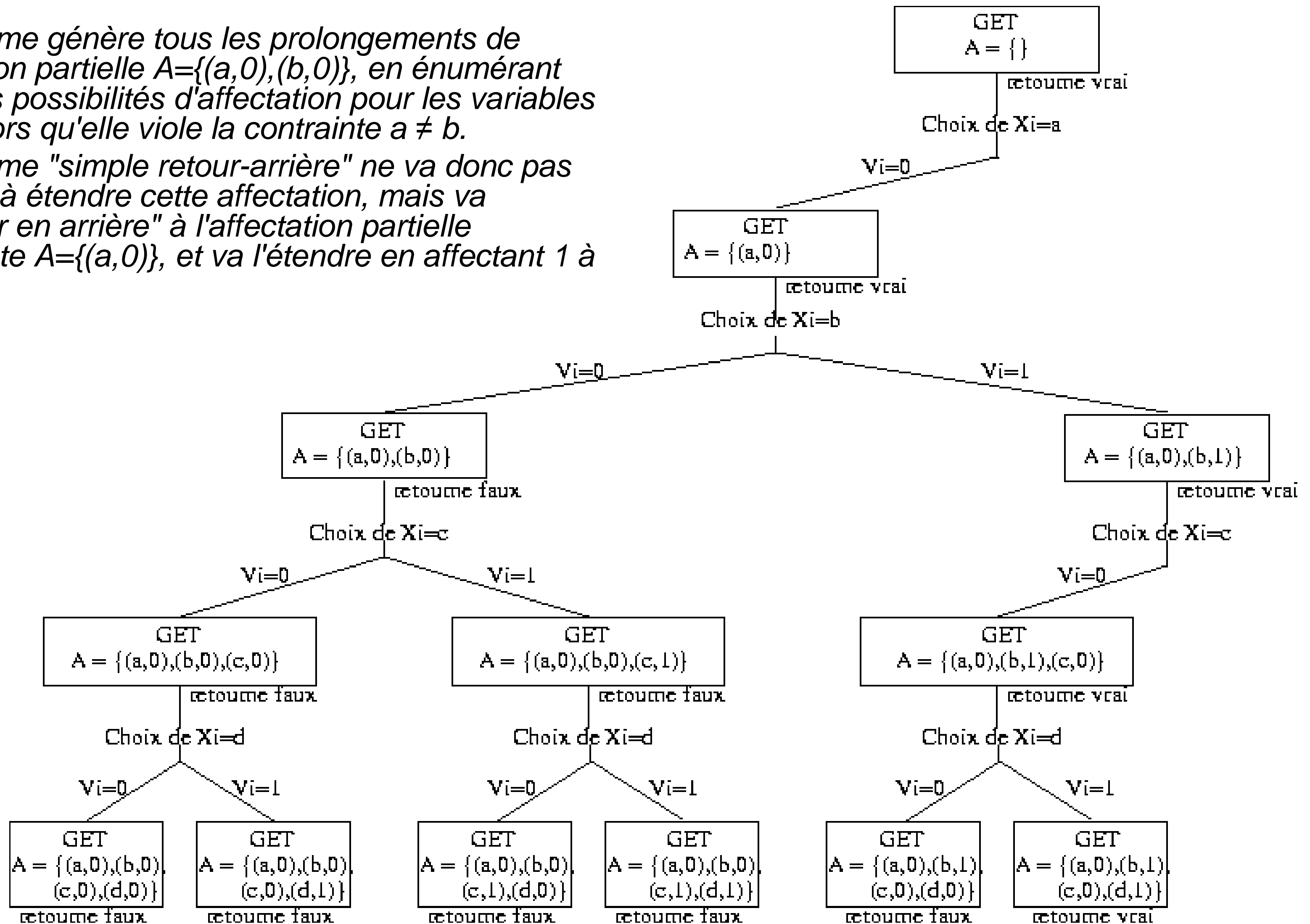
Une première façon d'améliorer l'algorithme "génère et teste" consiste à tester au fur et à mesure de la construction de l'affectation partielle sa consistance : dès lors qu'une affectation partielle est inconsistante, il est inutile de chercher à la compléter.

→ Dans ce cas, on "retourne en arrière" ("backtrack" en anglais) jusqu'à la plus récente instanciation partielle consistante que l'on peut étendre en affectant une autre valeur à la dernière variable affectée.

Exemple

L'algorithme génère tous les prolongements de l'affectation partielle A={(a,0),(b,0)}, en énumérant toutes les possibilités d'affectation pour les variables c et d, alors qu'elle viole la contrainte a ≠ b.

L'algorithme "simple retour-arrière" ne va donc pas chercher à étendre cette affectation, mais va "retourner en arrière" à l'affectation partielle précédente A={(a,0)}, et va l'étendre en affectant 1 à b , ...



Principe

`simpleRetourArrière(A,(X,D,C))`

Dans cette fonction, *A* contient une affectation partielle et *(X,D,C)* décrit le CSP à résoudre (au premier appel de cette fonction, l'affectation partielle *A* sera vide).

La fonction retourne `vrai` si on peut étendre l'affectation partielle *A* en une affectation totale consistante (une solution), et `faux` sinon.

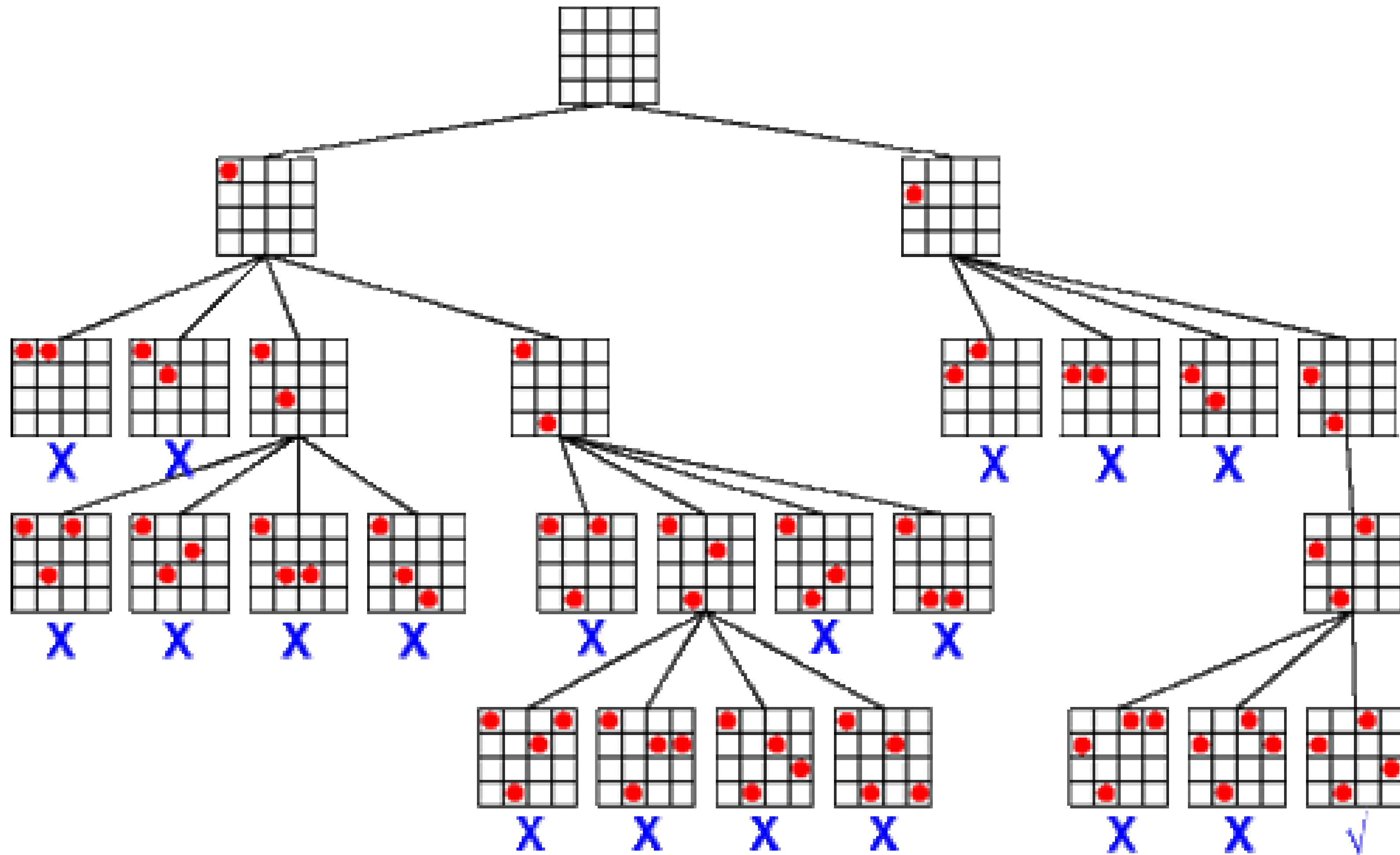
Algorithme

```
Fonction simpleRetourArrière(A,(X,D,C)) retourne un
   booléen
   Précondition :        A = affectation partielle
                 (X,D,C) = un CSP sur les domaines finis
   Postrelation :
   retourne vrai si A peut être étendue en une solution pour (X,D,C), faux
   sinon

début
   si A n'est pas consistante alors retourner faux finsi
   si toutes les variables de X sont affectées à une valeur dans A alors
   /* A est une affectation totale et consistante = une solution */
        retourner vrai
sinon /* A est une affectation partielle consistante */
   choisir une variable Xi de X qui n'est pas encore affectée à une valeur
   dans A
   pour toute valeur Vi appartenant à D(Xi) faire
   si simpleRetourArrière(A U {(Xi,Vi)}, (X,D,C)) = vrai alors retourner
   vrai

finpour
   retourner faux

finsi

fin
```

Exemple : Problème des *n* reines

- Variables : $X = \{X_1, X_2, X_3, X_4\}$
- Domaines : $D(X_1) = D(X_2) = D(X_3) = D(X_4) = \{1,2,3,4\}$
- Contraintes : $C = \{X_i \neq X_j$ / *i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j*$\}$
  U $\{X_i + i \neq X_j + j$ / *i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j*$\}$
  U $\{X_i - i \neq X_j - j$ / *i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j*$\}$

Exemple d'exécution



4- Algorithme « Look-Ahead »

Filtrage et consistance locale 1/

Pour améliorer l'algorithme "simple retour-arrière", on peut tenter d'anticiper ("look ahead" en anglais) les conséquences de l'affectation partielle en cours de construction sur les domaines des variables qui ne sont pas encore affectées :

→ Si on se rend compte qu'une variable non affectée X_i n'a plus de valeur dans son domaine $D(X_i)$ qui soit "localement consistante" avec l'affectation partielle en cours de construction, alors il n'est pas nécessaire de continuer à développer cette branche, et on peut tout de suite retourner en arrière pour explorer d'autres possibilités.

Filtrage et consistance locale 2/

Pour mettre ce principe en oeuvre, on va, à chaque étape de la recherche, filtrer les domaines des variables non affectées en enlevant les valeurs "localement inconsistantes", c'est-à-dire celles dont on peut inférer qu'elles n'appartiendront à aucune solution. On peut effectuer différents filtrages, correspondant à différents niveaux de consistances locales, qui vont réduire plus ou moins les domaines des variables, mais qui prendront aussi plus ou moins de temps à s'exécuter.

Consistance de nœud

Considérons un CSP *(X,D,C)*, et une affectation partielle consistante *A*, le filtrage le plus simple consiste à anticiper d'une étape l'énumération : pour chaque variable X_i non affectée dans *A*, on enlève de $D(X_i)$ toute valeur *v* telle que l'affectation $A \cup \{(X_i, v)\}$ soit inconsistante.

Par exemple pour le problème des 4 reines, après avoir instancié X_1 à 1, on peut enlever du domaine de X_2 la valeur 1 (qui viole la contrainte $X_1 \neq X_2$) et la valeur 2 (qui viole la contrainte $1-X_1 \neq 2-X_2$).

Ce filtrage permet d'établir ce qu'on appelle la ***consistance de noeud***, aussi appelée ***1-consistance***.

Définition formelle

- Un CSP *(X,D,C)* est consistant de noeud si pour toute variable X_i de *X*, et pour toute valeur *v* de $D(X_i)$, l'affectation partielle $\{(X_i,v)\}$ satisfait toutes les contraintes unaires de *C*.

  Par exemple, si *C* contient la contrainte *"X1 > 2"*, et si le domaine de X_1 contient les valeurs *{1,2,3,4,5}*, alors le CSP n'est pas consistant de noeud. Pour qu'il soit consistant de noeud, il faut enlever du domaine de X_1 les valeurs *1* et *2* qui violent la contrainte "$X_1 > 2$".

Consistance d'arc

Un filtrage plus fort, mais aussi plus long à effectuer, consiste à anticiper de deux étapes l'énumération : pour chaque variable X_i non affectée dans *A*, on enlève de $D(X_i)$ toute valeur *v* telle qu'il existe une variable X_j non affectée pour laquelle, pour toute valeur *w* de $D(X_j)$, l'affectation $A \cup \{(X_i,v),(X_j,w)\}$ soit inconsistante.

Par exemple pour le problème des 4 reines, après avoir instancié X_1 à 1, on peut enlever la valeur 3 du domaine de X_2 car si $X_1=1$ et $X_2=3$, alors la variable X_3 ne peut plus prendre de valeurs : si $X_3=1$, on viole la contrainte $X_3 \neq X_1$; si $X_3=2$, on viole la contrainte $X_3+3 \neq X_2+2$; si $X_3=3$, on viole la contrainte $X_3 \neq X_2$; et si $X_3=4$, on viole la contrainte $X_3-3 \neq X_2-2$. Notons que ce filtrage doit être répété jusqu'à ce plus aucun domaine ne puisse être réduit.

Ce filtrage permet d'établir ce qu'on appelle la ***consistance d'arc***, aussi appelée ***2-consistance***.

Définition formelle

Définition de la consistance d'arc : Un CSP *(X,D,C)* est consistant d'arc si pour tout couple de variables (X_i,X_j) de *X*, et pour toute valeur v_i appartenant à $D(X_i)$, il existe une valeur v_j appartenant à $D(X_j)$ telle que l'affectation partielle $\{(X_i,v_i),(X_j,v_j)\}$ satisfasse toutes les contraintes binaires de *C*.

Par exemple, si *C* contient la contrainte $"X_1 + X_2 > 2"$, et si $D(X_1)=D(X_2)=\{0,1,2\}$, alors le CSP n'est pas consistant d'arc car lorsque $X_1=0$, il n'y a aucune valeur de $D(X_2)$ qui puisse satisfaire la contrainte $"X_1 + X_2 > 2"$. Pour qu'il soit consistant d'arc, il faut enlever des domaines de X_1 et X_2 la valeur *0*.

L'algorithme qui enlève les valeurs des domaines des variables d'un CSP jusqu'à ce qu'il soit consistant d'arc (on dit que l'algorithme filtre les domaines des variables) s'appelle AC (pour Arc Consistency). Il existe différentes versions de cet algorithme: AC1, AC2, AC3, ..., chaque version étant plus efficace (en général) que la précédente.

Consistence de chemin…

Un filtrage encore plus fort, mais aussi encore plus long à effectuer, consiste à anticiper de trois étapes l'énumération. Ce filtrage permet d'établir ce qu'on appelle la ***consistance de chemin***, aussi appelée ***3-consistance***.

... et ainsi de suite... notons que s'il reste *k* variables à affecter, et si l'on anticipe de *k* étapes l'énumération pour établir la k-consistance, l'opération de filtrage revient à résoudre le CSP, c'est-à-dire que toutes les valeurs restant dans les domaines des variables après un tel filtrage appartiennent à une solution.

Principe de l'algorithme « anticipation »

Le principe général de l'algorithme "anticipation" reprend celui de l'algorithme "simple retour-arrière", en ajoutant simplement une étape de filtrage à chaque fois qu'une valeur est affectée à une variable.

Comme on vient de le voir, on peut effectuer différents filtrages plus ou moins forts, permettant d'établir différents niveaux de consistance locale (noeud, arc, chemin, ...).

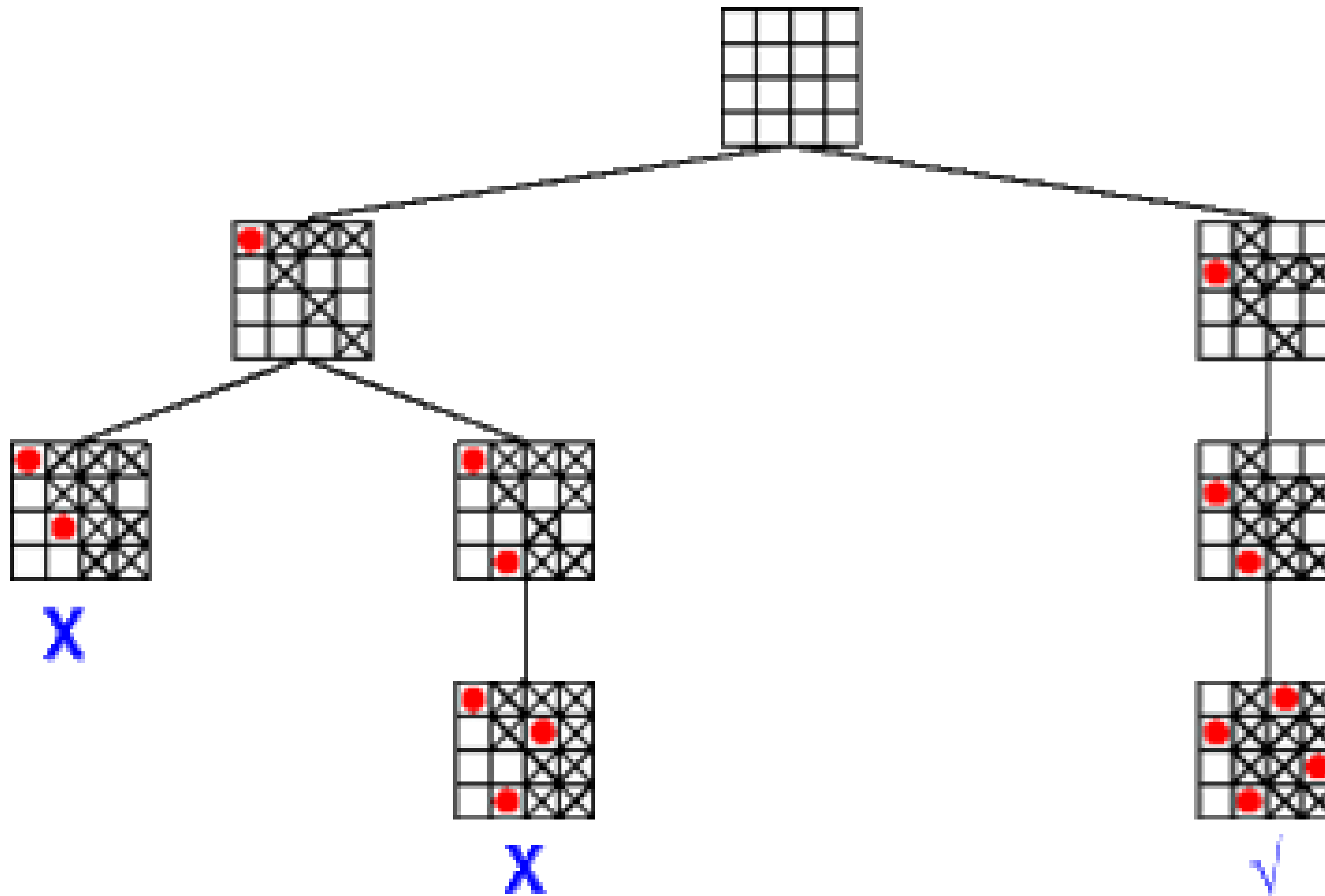
On va considérer par exemple un filtrage simple qui établit à chaque étape la consistance de noeud.

Dans cette fonction, *A* contient une affectation partielle consistante et *(X,D,C)* décrit le CSP à résoudre (au premier appel de cette fonction, l'affectation partielle *A* sera vide). La fonction retourne *vrai* si on peut étendre l'affectation partielle *A* en une affectation totale consistante (une solution), et faux sinon.

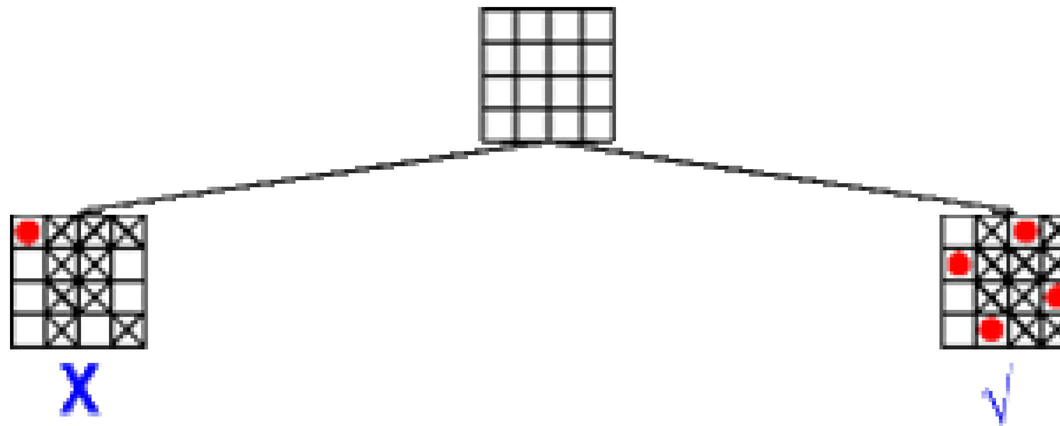
Algorithme

```
fonction anticipation/noeud(A,(X,D,C)) retourne un booléen
   Précondition :
   A = affectation partielle consistante
   (X,D,C) = un CSP sur les domaines finis

Postrelation :
   retourne vrai si A peut être étendue en une solution pour (X,D,C), faux
   sinon

Début
si toutes les variables de X sont affectées à une valeur dans A alors
   /* A est une affectation totale et consistante = une solution */
   retourner vrai
sinon /* A est une affectation partielle consistante */
choisir une var. Xi de X qui n'est pas encore affectée à une valeur dans A
   pour toute valeur Vi appartenant à D(Xi) faire
   /* filtrage des domaines par rapport à A U {(Xi,Vi)} */
        pour toute variable Xj de X qui n'est pas encore affectée faire
                Dfiltré(Xj) ← { Vj in D(Xj)|A U {(Xi,Vi),(Xj,Vj)} est consistant}
                si Dfiltré(Xj) est vide alors retourner faux
        finpour
   si anticipation(A U {(Xi,Vi)}, (X,Dfiltré,C))=vrai alors retourner vrai
   finpour
   retourner faux
finsi
fin
```

Filtrage « nœud »



Filtrage « arc »



Analyse 1/2

On constate sur le problème des 4 reines que le filtrage des domaines permet de réduire le nombre d'appels récursifs : on passe de 27 appels pour "simple retour-arrière" à 8 appels pour l'algorithme d'anticipation avec filtrage simple établissant une consistance de noeud.

En utilisant des filtrages plus forts, comme celui qui établit la consistance d'arc, on peut encore réduire la combinatoire de 8 à 3 appels récursifs.

Cependant, il faut noter que plus le filtrage utilisé est fort, plus cela prendra de temps pour l'exécuter...

Analyse 2/2

De façon générale, on constate que, quel que soit le problème considéré, **l'algorithme "anticipation/noeud" est généralement plus rapide que l'algorithme "simple retour-arrière"** car le filtrage utilisé est vraiment peu coûteux.

En revanche, si l'algorithme "anticipation/arc" envisage toujours moins de combinaisons que l'algorithme "anticipation/noeud", **il peut parfois prendre plus de temps à l'exécution car le filtrage pour établir une consistance d'arc est plus long que celui pour établir la consistance de noeud**.

5- Heuristiques

Heuristique, kézako ?

Les algorithmes que nous venons d'étudier choisissent, à chaque étape, la prochaine variable à instancier parmi l'ensemble des variables qui ne sont pas encore instanciées ; ensuite, une fois la variable choisie, ils essayent de l'instancier avec les différentes valeurs de son domaine.

→ Ces algorithmes ne disent rien sur l'ordre dans lequel on doit instancier les variables, ni sur l'ordre dans lequel on doit affecter les valeurs aux variables.

/!\ Ces deux ordres peuvent changer considérablement l'efficacité de ces algorithmes : imaginons qu'à chaque étape on dispose des conseils d'un "oracle-qui-sait-tout" qui nous dise quelle valeur choisir sans jamais se tromper ; dans ce cas, la solution serait trouvée sans jamais retourner en arrière...

Malheureusement, le problème général de la satisfaction d'un CSP sur les domaines finis étant NP-complet, il est plus qu'improbable que cet oracle fiable à 100% puisse jamais être "programmé".

En revanche, on peut intégrer des heuristiques pour déterminer l'ordre dans lequel les variables et les valeurs doivent être considérées :

Une heuristique est une règle non systématique (dans le sens où elle n'est pas fiable à 100%) qui nous donne des indications sur la direction à prendre dans l'arbre de recherche.

Idée générale

Les heuristiques concernant l'ordre d'instanciation des valeurs sont généralement dépendantes de l'application considérée et difficilement généralisables.

En revanche, il existe de nombreuses heuristiques d'ordre d'instanciation des variables qui permettent bien souvent d'accélérer considérablement la recherche.

L'idée générale consiste à instancier en premier les variables les plus "critiques", c'est-à-dire celles qui interviennent dans beaucoup de contraintes et/ou qui ne peuvent prendre que très peu de valeurs.

Exemples

L'ordre d'instanciation des variables peut être :

- ***statique***, quand il est fixé avant de commencer la recherche ;

  Par exemple, on peut ordonner les variables en fonction du nombre de contraintes portant sur elles : l'idée est d'instancier en premier les variables les plus contraintes, c'est-à-dire celles qui participent au plus grand nombre de contraintes.

- ou ***dynamique***, quand la prochaine variable à instancier est choisie dynamiquement à chaque étape de la recherche.

  Par exemple, l'heuristique "échec d'abord" ("first-fail" en anglais) consiste à choisir, à chaque étape, la variable dont le domaine a le plus petit nombre de valeurs localement consistantes avec l'affectation partielle en cours. Cette heuristique est généralement couplée avec l'algorithme "anticipation", qui filtre les domaines des variables à chaque étape de la recherche pour ne garder que les valeurs qui satisfont un certain niveau de consistance locale.