

Contraintes sur un domaine fini

- On a pour toute variable un *choix fini* de ses valeurs.
- Une classe importante de domaine de contraintes.
- Utilisée pour modéliser des problèmes avec des choix.
- Ordonnancement, Emploi du temps, routage, etc.
- Beaucoup d'applications industrielles.

Plan de ce cours

Contraintes sur un domaine fini.

- Solutionneur “gènère et teste”
- Solutionneur par retour en arrière
- Consistance d’arc et de nœuds
- Heuristiques
- Consistance de bornes
- Consistance généralisée

Dans ce cours on parle seulement du *solutionneur*. Dans le cours suivant on parlera de la programmation.

Problème de satisfaction de contraintes

- Une contrainte C sur des variables x_1, \dots, x_n
- Un domaine $D(x_i)$ pour chaque variable
- Une contrainte C est implicitement donnée par

$$C \wedge x_1 \in D(x_1) \wedge \dots \wedge x_n \in D(x_n)$$

- Contrainte *binnaire* : Ses contraintes simples contiennent au plus deux variables. Donne lieu à un *graphe de contraintes*.
- On écrit $\ll D(x) = \{c_1, \dots, c_n\} \gg$ au lieu de $\ll x \in \{c_1, \dots, c_n\} \gg$.

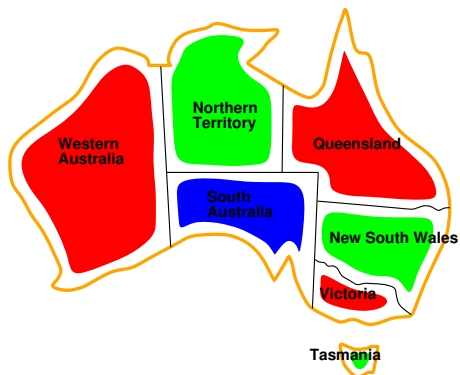
Exemple: Colorer une carte

Il y a trois couleurs. Des régions adjacentes doivent avoir des couleurs différentes.

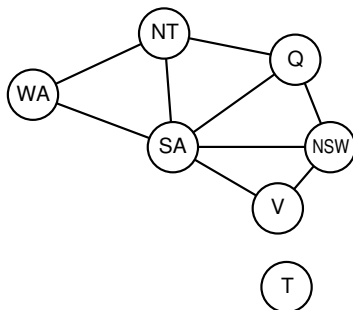


$$\begin{aligned} &WA \neq NT \wedge WA \neq SA \wedge NT \neq SA \wedge NT \neq Q \wedge SA \neq Q \wedge SA \neq V \\ &\wedge NSW \neq SA \wedge NSW \neq Q \wedge NSW \neq V \\ &D(WA) = D(NT) = D(SA) = D(Q) = D(V) = D(NSW) = D(T) = \\ &\{rouge, jaune, bleu\} \end{aligned}$$

Exemple: Colorer une carte



Graphe de contraintes



Exemple: Les 4 reines

- Placer 4 reines sur un échiquier de taille 4x4 de sorte qu'aucune reine est en prise
- Quatre variables Q_1, Q_2, Q_3, Q_4 qui représentent la ligne de la reine dans chaque colonne. Domaine de chaque variable: $\{1, 2, 3, 4\}$
- Les contraintes: $Q_1 \neq Q_2 \wedge Q_1 \neq Q_3 \wedge Q_1 \neq Q_4 \wedge$
 $Q_2 \neq Q_3 \wedge Q_2 \neq Q_4 \wedge Q_3 \neq Q_4$
- $Q_1 \neq Q_2 + 1 \wedge Q_1 \neq Q_3 + 2 \wedge Q_1 \neq Q_4 + 3 \wedge$
 $Q_2 \neq Q_3 + 1 \wedge Q_2 \neq Q_4 + 2 \wedge Q_3 \neq Q_4 + 1$
- $Q_1 \neq Q_2 - 1 \wedge Q_1 \neq Q_3 - 2 \wedge Q_1 \neq Q_4 - 3 \wedge$
 $Q_2 \neq Q_3 - 1 \wedge Q_2 \neq Q_4 - 2 \wedge Q_3 \neq Q_4 - 1$

Exemple: sac du contrebandier

- Contrebandier avec un sac de capacité 9.
- Il doit choisir des objets pour faire un profit d'au moins 30

- | objet | profit | poids |
|------------|--------|-------|
| whisky | 15 | 4 |
| parfum | 10 | 3 |
| cigarettes | 7 | 2 |

$$4W + 3P + 2C \leq 9 \wedge 15W + 10P + 7C \geq 30$$

- Domaines des variables ?

Solutionneur génère et teste

- Le plus simple est d'énumérer les affectations possibles.
- Le solutionneur **génère et teste** :
 - ▶ Énumère une par une les valeurs des variables une par une
 - ▶ Quand *chaque* variable a une valeur, on teste si la contrainte est satisfaite ou pas.
- Très inefficace !
- On peut améliorer cette technique en testant à chaque fois, si l'affectation partielle entraîne déjà la non-satisfaisabilité : voir le transparent suivant.

Solutionneur simple par retour en arrière

- Le solutionneur simple par retour en arrière:
 - ▶ énumère une par une les valeurs des variables une par une
 - ▶ vérifie qu'aucune contrainte simple est fautive à chaque étape
 - ▶ On peut facilement tester la satisfaisabilité d'une contrainte simple sans variables.
 - ▶ $partsat(C)$ retourne faux, si C n'est pas satisfaisable à cause d'une contrainte simple sans variables (close) qui n'est pas satisfaisable. Sinon $partsat(C)$ retourne vrai.

Solutionneur par retour en arrière

$partsat(C) = vrai$ ssi toute contrainte simple et close de C est vraie

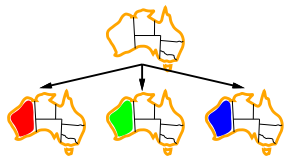
$backsolve(C, D)$

- Si $variables(C)$ est vide, alors retourne $partsat(C)$
- Choisir x dans $variables(C)$
- Pour chaque valeur d dans $D(x)$
 - ▶ Soit C_1 la contrainte C où x est remplacé par d
 - ▶ Si $partsat(C_1)$ alors
 si $backsolve(C_1, D)$ alors retourne vraie
- Retourne *faux*

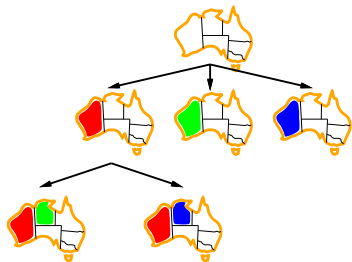
Exemple retour en arrière



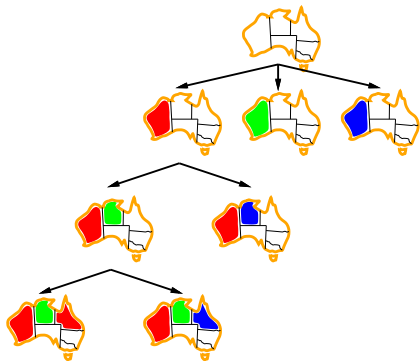
Exemple retour en arrière



Exemple retour en arrière



Exemple retour en arrière



Complexité des contraintes sur un domaine fini

- Satisfaisabilité de contraintes sur un domaine fini est NP-complet (même si on se restreint à des domaines de cardinalité 2, p.ex. satisfaisabilité de clauses propositionnelles: 3SAT).
- C'est-à-dire :
 - ▶ On ne connaît que des algorithmes avec complexité exponentielle au pire des cas.
 - ▶ Il est “fort probable” qu'il n'y a pas d'algorithme avec une meilleure complexité.
- Voir un cours de *algorithmique* ou *calculabilité et complexité*.

Consistance de nœud et d'arc

- Idée: Trouver un CSP équivalent au CSP d'origine qui a des domaines de variables plus petits.
- C'est le principe de la *simplification de contraintes*, appliqué aux domaines des variables.
- **On considère les contraintes simples une par une.**
- Consistance de nœud : ($variables(c) = \{x\}$): enlever chaque valeur du domaine de x qui rend la contrainte simple c insatisfaisable.
- Consistance d'arc : ($variables(c) = \{x, y\}$): enlever chaque valeur de $D(x)$ pour laquelle il n'y a pas de valeur dans $D(y)$ qui satisfait la contrainte simple c et vice-versa.

Consistance de nœud

- Une contrainte simple c est nœud-consistante avec domaine D , si
 - ▶ soit $|variables(c)| \neq 1$;
 - ▶ soit $variables(c) = \{x\}$, et pour chaque d dans $D(x)$, $\{x \leftarrow d\}$ est une solution de c .
- Un CSP est nœud-consistant, si chaque contrainte simple est nœud-consistante.

Comment obtenir un CSP nœud-consistant ?

$nœudcons(C, D)$

- Pour chaque contrainte simple c dans C
 - ▶ $D := nœudconssimple(c, D)$
- retourne D

$nœudconssimple(c, D)$

- Si $|variables(c)| = 1$ alors
 - ▶ Soit $\{x\} = variables(c)$
 $D(x) := \{d \in D(x) \mid \{x \leftarrow d\} \text{ est une solution de } c\}$
- retourne D

Arc-consistance

- Une contrainte simple est arc-consistante avec domaine D , si
 - ▶ soit $|variables(c)| \neq 2$;
 - ▶ soit $variables(c) = \{x, y\}$ et
 - ★ pour chaque d dans $D(x)$, il y a $e \in D(y)$ tel que $\{x \leftarrow d, y \leftarrow e\}$ est une solution de c
 - ★ et analogue pour y .
- Un CSP est arc-consistant, si chaque contrainte simple est arc-consistante.

Comment obtenir un CSP arc-consistant ?

arcconssimple(c, D)

- si $|variables(c)| = 2$ alors
 - ▶ $D(x) := \{d \in D(x) \mid \exists e \in D(y) \text{ t.q. } \{x \leftarrow d, y \leftarrow e\}$
est une solution de $c\}$
 - ▶ $D(y) := \{e \in D(y) \mid \exists d \in D(x) \text{ t.q. } \{x \leftarrow d, y \leftarrow e\}$
est une solution de $c\}$
- retourne D

Enlève des valeurs non arc-consistantes avec c

Comment obtenir un CSP arc-consistant ?

arccons(C, D)

- Répète
 - ▶ $W := D$
 - ▶ Pour chaque contrainte simple c de C
 - ★ $D := \text{arcconssimple}(c, D)$
- jusqu'à $W = D$
- retourne D

Arc-consistance : calcul d'un point fixe

- Remarquer la boucle : le traitement d'une contrainte simple peut déclencher qu'une autre contrainte simple est à traiter de nouveau.
- Exemple:

$$X_1 < X_2 \wedge X_2 < X_3 \wedge X_3 < X_4$$

$$X_1, X_2, X_3, X_4 \in \{0, 1, 2, 3, 4\}$$

(sera fait au tableau)

Utiliser nœud et arc-consistance

- On peut définir des solveurs.
- Deux types de domaines importants :
 - ▶ domaine *faux* : une variable a un domaine vide
 - ▶ domaine *simple* : toutes les variables ont un domaine singleton (de taille un)
- On suppose qu'on a un test de satisfaisabilité ($satisfaisable(C, D)$) sur des CSPs avec des domaines simples.

Solutionneur nœud et arc-consistance

Solutionneur **incomplet**

- $D := \text{nœudcons}(C, D)$
- $D := \text{arccons}(C, D)$
- Si D est un domaine faux, alors retourne *faux*
- Si D est un domaine simple, alors retourne *satisfaisable*(C, D)
- sinon retourne *inconnu*

Comment définir un solutionneur **complet** en utilisant arc et nœud-consistance ?

Retour en arrière avec consistance

- Combiner le solveur par retour en arrière avec consistance.
- Appliquer nœud (et/ou) arc-consistance avant de lancer le solveur par retour en arrière **et** après chaque fois qu'une variable est affectée par le solveur.

Exemple avec nœud-consistance uniquement



Exemple avec nœud-consistance



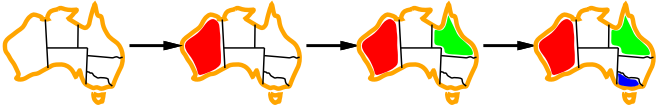
Exemple avec nœud-consistance



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

Ce problème est-il arc-consistant ?

Exemple avec nœud-consistance



WA

NT

Q

NSW

V

SA

T

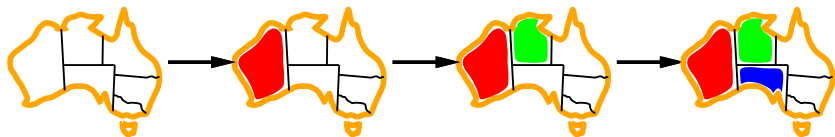
■	■	■	■	■	■	■	■	■	■	■	■
■	■	■	■	■	■	■	■	■	■	■	■
■	■	■	■	■	■	■	■	■	■	■	■
■	■	■	■	■	■	■	■	■	■	■	■

Heuristiques

- On peut utiliser des heuristiques pour choisir la variable à affecter et la valeur affectée.
- statique/dynamique

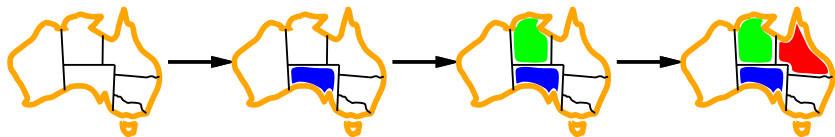
La variable la plus contrainte

- Choisir la variable avec le plus petit nombre de valeurs légales



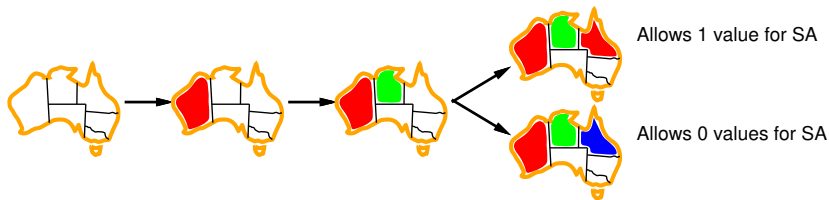
La variable la plus contraignante

- En cas d'égalité pour la variable la plus contrainte
- Choisir la variable qui a le plus de contraintes avec les variables qui restent



La valeur la moins contraignante

- Pour une variable, choisir la valeur qui contraint le moins possible les variables qui restent



Consistance pour des contraintes avec plus de 2 variables

- Quoi faire avec des contraintes avec plus de 2 variables ?
- Consistance d'hyper-arc: étendre l'arc-consistance à un nombre arbitraire de variables
- Déterminer la hyper-arc-consistance est NP-difficile

Consistance de bornes

- CSP arithmétique: les contraintes sont sur des entiers
- intervalles: $[l..u]$ représente l'ensemble $\{l, l + 1, \dots, u\}$
- Idée: Utiliser la consistance sur les réels et examiner seulement les bornes (inférieurs et supérieurs) du domaine de chaque variable
- Définir $\min(D, x)$ comme l'élément minimum dans le domaine de x , pareil $\max(D, x)$

Consistance de bornes

- Une contrainte simple c est bornes-consistante avec domaine D , si pour chaque variable x dans $variables(c)$
 - ▶ ils existent des réels d_1, \dots, d_k pour les autres variables x_1, \dots, x_k tel que
 - ★ $min(D, x_j) \leq d_j \leq max(D, x_j)$ pour tout j et
 - ★ $\{x \leftarrow min(D, x), x_1 \leftarrow d_1, \dots, x_k \leftarrow d_k\}$ est une solution de c
 - ▶ ils existent des réels d'_1, \dots, d'_k pour les autres variables x_1, \dots, x_k tel que
 - ★ $min(D, x_j) \leq d'_j \leq max(D, x_j)$ pour tout j et
 - ★ $\{x \leftarrow max(D, x), x_1 \leftarrow d'_1, \dots, x_k \leftarrow d'_k\}$ est une solution de c
- Un CSP arithmétique est bornes-consistant, si toutes ses contraintes simples le sont

Comment obtenir un CSP bornes-consistant ?

- Étant donné un domaine D , on doit modifier les bornes, de sorte que le résultat est bornes-consistant
- Utilisation de **règles de propagation**
- Exemple:
 - ▶ $X = Y + Z$ équivalent à $Y = X - Z$ et $Z = X - Y$
 - ▶ Raisonner avec *max* et *min*
 - ▶ $X \geq \min(D, Y) + \min(D, Z)$, $X \leq \max(D, Y) + \max(D, Z)$
 - ▶ $Y \geq \min(D, X) - \max(D, Z)$, $Y \leq \max(D, X) - \min(D, Z)$
 - ▶ $Z \geq \min(D, X) - \max(D, Y)$, $Z \leq \max(D, X) - \min(D, Y)$
 - ▶ permettent d'ajuster les domaines

Exemple

- $X = Y + Z$, $D(X) = [4..8]$, $D(Y) = [0..3]$, $D(Z) = [2..2]$
- Les règles de propagation donnent:
 - ▶ $(0 + 2 =) 2 \leq X \leq 5 (= 3 + 2)$
 - ▶ $(4 - 2 =) 2 \leq Y \leq 6 (= 8 - 2)$
 - ▶ $(4 - 3 =) 1 \leq Z \leq 8 (= 8 - 0)$
- Les domaines peuvent être réduits:
 $D(X) = [4..5]$, $D(Y) = [2..3]$, $D(Z) = [2..2]$

Autres règles de propagation

- $4W + 3P + 2C \leq 9$
- $W \leq \frac{9}{4} - \frac{3}{4}\min(D, P) - \frac{2}{4}\min(D, C)$
- $P \leq \frac{9}{3} - \frac{4}{3}\min(D, W) - \frac{2}{3}\min(D, C)$
- $C \leq \frac{9}{2} - \frac{4}{2}\min(D, W) - \frac{3}{2}\min(D, P)$
- Étant donné un domaine initial
 $D(W) = [0..9], D(P) = [0..9], D(C) = [0..9]$ on détermine que
 $W \leq \frac{9}{4}, P \leq \frac{9}{3}, C \leq \frac{9}{2},$
- nouveau domaine: $D(W) = [0..2], D(P) = [0..3], D(C) = [0..4]$

Inégalités $Y \neq Z$

- Contrairement à l'arc-consistance, Les inégalités donnent des règles de propagation très faibles
- cela est du au fait qu'on considère uniquement les bornes des interval
- Seulement si une de deux côtés prend une valeur fixe qui est égale au minimum ou maximum de l'autre il y a propagation
- $D(Y) = [2..4], D(Z) = [2..3]$ pas de propagation
- $D(Y) = [2..4], D(Z) = [3..3]$ pas de propagation
- $D(Y) = [2..4], D(Z) = [2..2]$ propagation
 $D(Y) = [3..4], D(Z) = [2..2]$

Algorithme de bornes consistance

- Rendre une **seule contrainte simple** borne consistante peut déjà entraîner plusieurs applications de règles de propagation
 - ▶ Exemple (au tableau):
 $3X = 2Y$ avec $D(X) = [0..5]$ et $D(Y) = [0..8]$
- $bornescons(C, D)$: Appliquer les règles de propagation pour chaque contrainte simple de C , jusqu'à ce qu'il n'y a plus de changement dans les domaines D .
- On ne réexamine pas une contrainte simple, si les domaines de ses variables n'ont pas changé.

Solutionneur bornes consistance

- $D := \text{bornescons}(C, D)$
- Si D est un domaine faux, alors retourne *false*
- Si D est un domaine simple, alors retourne *satisfaisable*(C, D)
- sinon retourne *inconnu*

Solutionneur par retour en arrière avec bornes consistance

- Appliquer bornes consistance avant de lancer le solutionneur par retour en arrière **et** à chaque fois qu'une variable est affectée par le solutionneur retour en arrière.

Exemple retour en arrière avec bornes consistance

- Problème du sac du contrebandier
- $4W + 3P + 2C \leq 9 \wedge 15W + 10P + 7C \geq 30$
- Domaines initiaux : $D(W) = [0..9]$, $D(P) = [0..9]$, $D(C) = [0..9]$
- Bornes consistance sur la première contrainte donne:
 $D(W) = [0..2]$, $D(P) = [0..3]$, $D(C) = [0..4]$
- On essaie $W = 0$. Ça donne : $D(W) = [0..0]$, $D(P) = [1..3]$,
 $D(C) = [0..3]$
- On essaie $P = 1$. Ça donne : $D(W) = [0..0]$, $D(P) = [1..1]$,
 $D(C) = [3..3]$ et on a trouvé une solution.
- On peut aussi chercher les autres solutions

Consistance généralisée

- On peut combiner les trois consistances (nœud, arc, bornes) vues jusqu'à présent.
- Toutes ses méthodes utilisent les contraintes simples une par une.
- On peut considérer des contraintes simples "complexes" qui sont une conjonction de contraintes simples avec un mécanisme de propagation spécial.
- Exemple: $alldifferent(\{V_1, \dots, V_n\})$
- $alldifferent(\{X, Y, Z\})$ signifie $X \neq Y \wedge Y \neq Z \wedge X \neq Z$
- Arc-consistant avec $D(X) = \{1, 2\}$, $D(Y) = \{1, 2\}$, $D(Z) = \{1, 2\}$
- Mais il n'y a pas de solution.

Consistance pour alldifferent

- Soit c de la forme $alldifferent(V)$
- Tant qu'il existe $v \in V$ avec $D(v) = \{d\}$
 - ▶ $V := V - \{v\}$
 - ▶ Pour chaque $v' \in V$
 - ★ $D(v') := D(v') - \{d\}$
- $DV := \bigcup_{v \in V} D(v)$
- Si $|V| > |DV|$ alors retourne *domaine faux*
- retourne D

Exemples alldifferent

- $alldifferent(\{X, Y, Z\})$ avec $D(X) = \{1, 2\}$, $D(Y) = \{1, 2\}$,
 $D(Z) = \{1, 2\}$
- L'algorithme retourne *faux*
- $alldifferent(\{X, Y, Z, T\})$ avec $D(X) = \{1, 2\}$, $D(Y) = \{1, 2\}$,
 $D(Z) = \{1, 2\}$, $D(T) = \{2, 3, 4, 5\}$
- L'algorithme ne détecte pas le problème
- On peut utiliser des algorithmes plus compliqués pour cela.

Exemple d'utilisation de alldifferent

		9			1	6	2	
5	7			2	8		3	
3			7					4
8	9			7		4		
	6		5		3		9	
		1		9			7	6
6					7			8
	4		1	3			6	5
	2	7	6			9		

Sudoku

- Le problème du Sudoku consiste à remplir une grille de sorte que chaque ligne, chaque colonne et chaque carré contiennent les chiffres 1 à 9.
- Pour modéliser ce problème on peut utiliser alldifferent. Comment ?

Les contraintes sur domaines finis en Eclipse CLP

- On utilise la librairie `ic` avec `lib(ic)`. (ou `:- lib(ic)`. dans le programme).
- Attention: cette librairie fait plus que les contraintes à domaines finis
- Nous utiliserons uniquement des variables entières.
- Leurs domaines doivent être définis.
par exemple `L = [X,Y]`, `L #:: [-3..10]`. ou `X #:: [1,2,5,6]`
- `alldifferent(L)` impose que toutes les variables de `L` ont des valeurs différentes.
- Les contraintes arithmétiques s'écrivent en utilisant les fonctions habituelles et les prédicats : `#=`, `#\=`, `#<`, `#>`, `#<=`, `#>=`.
Règles de propagation : bornes-consistance sauf pour `#\=` où l'arc-consistance est utilisée.
- `ac_eq(X,Y,C)` est l'implémentation avec arc-consistance de `X #= Y+C`.

Exemples

```
[eclipse]: X #:: [1..8], Y #:: [2..7], X # = 2*Y.
```

```
X = X{4 .. 8}
```

```
Y = Y{2 .. 4} ...
```

```
[eclipse]: [X,Y] #:: [0..9], X #\= 5, X # = Y+2.
```

```
X = X{[2 .. 4, 6 .. 9]}
```

```
Y = Y{0 .. 7} ...
```

```
[eclipse]: [X,Y] #:: [0..9], X #\= 5, ac_eq(X,Y,2).
```

```
X = X{[2 .. 4, 6 .. 9]}
```

```
Y = Y{[0 .. 2, 4 .. 7]} ...
```

Si on veut construire une expression et l'utiliser dans une contrainte à l'exécution, il faut l'encapsuler avec `eval/1`

```
[eclipse]: [X, Y] #:: 0..10, Expr = X + Y, Sum # = Expr.
```

```
number expected in set_up_ic_con(7, 1, [0 * 1, 1 *  
Sum{-1.0Inf .. 1.0Inf}, -1 * (X{0 .. 10} + Y{0 .. 10})])
```

```
[eclipse]: [X, Y] #:: 0..10, Expr = X + Y, Sum # = eval(Expr).
```

```
X = X{0 .. 10} Y = Y{0 .. 10}
```

```
Expr = X{0 .. 10} + Y{0 .. 10}
```

```
Sum = Sum{0 .. 20} ...
```

Chercher une solution

- `indomain(+Dvar)`
instantie la variable `Dvar` avec une valeur de son domaine, en revenant par retour en arrière une nouvelle valeur est prise.
- `labeling(?Vars)`
Ce prédicat est utilisé pour rechercher des solutions de contraintes sur les variables `Vars`. Il utilise `indomain`.
- `search(+L, ++Arg, ++Select, +Choice, ++Method, +Option)`
permet de chercher une solution pour les variables `L` avec plein d'options (voir <http://eclipseclp.org/doc/bips/lib/ic/search-6.html>).
Typiquement `Arg` est `0`, `Select` est `first_fail`, `Choice` est `indomain_min`, `Method` est `complete` et `Option` est `[]`.

Structure générale d'un programme simple

- Dans le cas le plus simple, un programme pour résoudre une contrainte s'écrit en trois parties :
 - ▶ définir les domaines des variables
 - ▶ décrire la contrainte
 - ▶ labeling ou search
- Exemple:

```
probleme([X,Y,Z]) :- X #:: [0..5],  
                    [Y,Z] #:: [3..7],  
                    X+Y #< 2*Z,  
                    labeling([X,Y,Z]).
```

```
[eclipse]: probleme(L).
```

```
L = [0,3,3] ? ;
```

```
L = [0,3,4] ? ;
```

```
L = [0,3,5] ? ;
```

```
...
```

Exemple: sac du contrebandier

- Contrebandier avec un sac de capacité 9.
- Il doit choisir des objets pour faire un profit d'au moins 30

objet	profit	poids
whisky	15	4
parfum	10	3
cigarettes	7	2

$$4W + 3P + 2C \leq 9 \wedge 15W + 10P + 7C \geq 30$$

Le contrebandier en Eclipse CLP

```
[eclipse]: [W,P,C] #:: [0..9],  
    4*W + 3*P + 2*C #=< 9,  
    15*W + 10*P + 7*C #>= 30,  
    labeling([W,P,C]).
```

W = 0

P = 1

C = 3 ? ;

W = 0

P = 3

C = 0 ? ;

W = 1

P = 1

C = 1 ? ;

W = 2

P = 0

C = 0

Éliminer des symétries

- Des symétries dans l'arbre de recherche peuvent faire un programme naïf très inefficace.
- On peut avoir beaucoup de façons symétriques d'écrire une partie de l'arbre de recherche.
- C'est un problème quand il s'agit d'une partie de l'arbre de recherche dont tous les nœuds sont des nœuds d'échec.
- Si on élimine des symétries : attention quand on essaye de trouver *toutes* les solutions.

Exemple : élimination de symétries

- Dans un magasin : le client veut payer quatre articles.
- Le caissier lui annonce un prix de 7.11 €.
- Quand le client lui demande les prix des articles séparés, le caissier lui répond simplement que le produit des prix est également 7.11 €.
- Quels sont les prix des quatre articles ?

Exemple 7.11 € en ECLIPSE CLP

```
prix(A,B,C,D) :-  
    [A,B,C,D] #:: [1..708],  
    A+B+C+D #= 711,  
    A*B*C*D #= 711 * 100 * 100 * 100,  
    labeling([A,B,C,D]).
```

```
[eclipse] : prix(A,B,C,D).
```

A = 120

B = 125

C = 150

D = 316 ?

Yes (25.70s cpu,...)

Est-ce qu'il y a d'autres solutions ?

Éliminer les symétries

Idée : on impose l'ordre sur les valeurs des variables.

```
prixa(A,B,C,D) :-  
    [A,B,C,D]#::[1..708],  
    A+B+C+D #= 711,  
    A*B*C*D #= 711 * 100 * 100 * 100,  
    A #=< B, B #=< C, C #=< D,  
    labeling([A,B,C,D]).
```

Trouve **la** solution en 3.47 secondes.

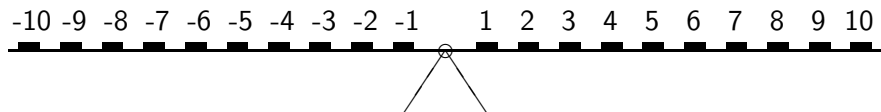
Encore mieux

Idée : 79 est un facteur premier de 711. Choisir la variable qui contient ce facteur.

```
prixb(A,B,C,D) :-  
    [A,B,C,D]#::[1..708],  
    A+B+C+D #= 711,  
    A*B*C*D #= 711 * 100 * 100 * 100,  
    A #= 79*X,  
    B #=< C, C #=< D,  
    labeling([A,B,C,D]).
```

Trouve la solution en 0.62 secondes.

Exemple: Programmer la génération des contraintes



- Donné : liste des poids des personnes.
- Exemple : $[30, 40]$
- Placer toutes les personnes de sorte que la balançoire soit en équilibre.
- Pour l'exemple : $[-4, 3]$.

Solution balançoire

```
placement(Poids,Places) :-  
    length(Poids,N),  
    length(Places,N),  
    Places #:: [-10..10],  
    alldifferent([0|Places]),  
    gauche(Poids,Places,M),  
    droite(Poids,Places,M),  
    labeling(Places).
```

Solution balanceoire

```
gauche([], [], 0).
gauche([_ | Poids], [Place | Places], M) :-
    Place #> 0, gauche(Poids, Places, M).
gauche([Poid | Poids], [Place | Places], M) :-
    Place #< 0,
    M #= MM - Poid * Place,
    gauche(Poids, Places, MM).
droite([], [], 0).
droite([_ | Poids], [Place | Places], M) :-
    Place #< 0, droite(Poids, Places, M).
droite([Poid | Poids], [Place | Places], M) :-
    Place #> 0,
    M #= MM + Poid * Place,
    droite(Poids, Places, MM).
```


Exemple: Pavage

- Donnée: une liste de pavés (exemple: $[(2, 2), (3, 4)]$) et les dimensions du plan.
- Déterminer si on peut placer les pavés sur le plan sans recouvrement, et si oui donner le plan (les coordonnées des coins des pavés placés).
- Ici : utiliser une contrainte définie par l'utilisateur qui exprime que deux pavés ne se recouvrent pas.
- Ici pour simplifier : on n'autorise pas de tourner les pavés.
- La position du pavé est donnée par quatre valeurs (les coordonnées du point le plus à gauche et le plus bas et les coordonnées du coin opposé).

Exemple : pavage (1)

```
disjoint( (_,_,X2,_), (X1,_,_,_) ) :- X2 #=< X1.
```

```
disjoint( (X1,_,_,_), (_,_,X2,_) ) :- X1 #>= X2.
```

```
disjoint( (_,_,_,Y2), (_,Y1,_,_) ) :- Y2 #=< Y1.
```

```
disjoint( (_,Y1,_,_), (_,_,_,Y2) ) :- Y1 #>= Y2.
```

```
alldisjoint( _, [] ).
```

```
alldisjoint( Q, [H|T] ) :- disjoint( Q, H ), alldisjoint( Q, T ).
```

Exemple : pavage (2)

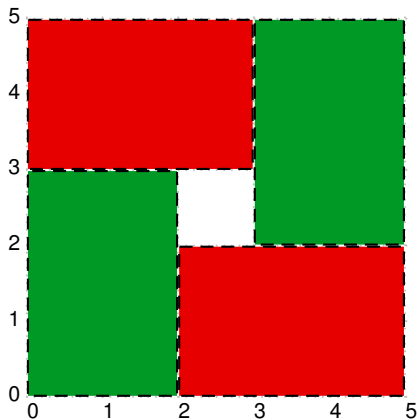
```
p([],_,-, [], []).  
p([(X,Y)|Rest], Width, Height, [(X1,Y1,X2,Y2)|Plan],  
  [X1,Y1,X2,Y2|Vars]) :-  
    [X1,X2]#::[0..Width],  
    [Y1,Y2]#::[0..Height],  
    X2 #= X1 + X,  
    Y2 #= Y1 + Y,  
    p(Rest,Width, Height, Plan, Vars),  
    alldisjoint((X1,Y1,X2,Y2), Plan).  
  
paving(In,Width,Height,Plan) :-  
    p(In,Width,Height,Plan,Vars),  
    labeling(Vars).
```

Exemple : le problème de la guillotine

- On veut couper une feuille de papier en plusieurs morceaux.
- On a seulement une guillotine pour découper la feuille.
- Étant donné les dimensions de la feuille et des morceaux, est-ce possible ?



Différence avec le problème de pavage



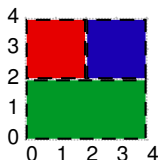
Spécification du problème

Écrire un prédicat coupe(?Pieces,?Hauteur,?Largeur,+Plan) où

- Pieces est la liste des dimensions des morceaux souhaités,
- Largeur et Hauteur sont les dimensions du plan à découper
- Plan est le plan de découpage.

```
| : coupe([(2,2),(2,2),(2,4)],4,4,P).
```

```
P = horizontal(2,vertical(2,piece(2,2),piece(2,2)),  
              piece(4,2))
```



L'idée de l'algorithme

- Construction d'un arbre.
- Si on veut un seul morceaux, et s'il tient dans les dimensions : créer une feuille de l'arbre.
- Si on veut plusieurs morceaux :
 - ▶ choisir la ligne de découpage.
 - ▶ choisir une partition des morceaux telle que les sommes de surfaces tiennent dans les deux moitiés du plan.
 - ▶ appeler coupe récursivement sur les deux sous-problèmes, combiner les plans obtenus pour construire le plan entier.

Première solution naïve

```
coupe([(X,Y)],L,H,piece(X,Y)) :- X #=< L, Y #=< H.
```

```
coupe([(X,Y)],L,H,piece(Y,X)) :- Y #=< L, X #=< H.
```

```
coupe(Pieces,Largeur,Hauteur,vertical(Coupe,PlanA,PlanB)) :-
```

```
  Coupe#::[1..Largeur],
```

```
  RestLargeur #= Largeur-Coupe,
```

```
  SurfaceA #= Coupe*Hauteur,
```

```
  SurfaceB #= RestLargeur*Hauteur,
```

```
  partition(Pieces,ResultatA,ResultatB,SurfaceA,SurfaceB),
```

```
  labeling([Coupe]),
```

```
  coupe(ResultatA,Coupe,Hauteur,PlanA),
```

```
  coupe(ResultatB,RestLargeur,Hauteur,PlanB).
```

```
coupe(Pieces,Largeur,Hauteur,horizontal(Coupe,PlanA,PlanB)) :-
```

```
  ...
```


Solution naïve suite

```
partition([], [], [], _, _).
```

```
partition([(X,Y)|T], [(X,Y)|R], ResultatB, SurfaceA, SurfaceB) :-  
    X*Y #=< SurfaceA,  
    SurfaceA #= X*Y + NewSurfaceA,  
    partition(T, R, ResultatB, NewSurfaceA, SurfaceB).
```

```
partition([(X,Y)|T], ResultatA, [(X,Y)|R], SurfaceA, SurfaceB) :-  
    X*Y #=< SurfaceB,  
    SurfaceB #= X*Y + NewSurfaceB,  
    partition(T, ResultatA, R, SurfaceA, NewSurfaceB).
```

Testons la solution naïve

```
[eclipse] : coupe([(2,2),(2,2),(2,4)],4,3,P).
```

No

```
[eclipse] : coupe([(2,2),(2,2),(2,4)],4,4,P).
```

Stack Overflow

L'erreur de la solution naïve

- La récurrence ne s'arrête pas :
 - ▶ On permet de couper tel qu'un plan a la surface 0.
 - ▶ On permet de partitionner la liste vide en deux listes vides.
- Deux solutions possibles :
 - ▶ Renforcer la contrainte sur la variable Coupe
 - ▶ Ne pas permettre des partitions triviales (une liste vide).

Le programme corrigé

```
coupe([(X,Y)],L,H,piece(X,Y)) :- X #=< L, Y #=< H.
```

```
coupe([(X,Y)],L,H,piece(Y,X)) :- Y #=< L, X #=< H.
```

```
coupe(Pieces,Largeur,Hauteur,vertical(Coupe,PlanA,PlanB)) :-  
  Coupe #:: [1..Largeur],
```

```
  Coupe#<Largeur,
```

```
  RestLargeur #= Largeur-Coupe,
```

```
  SurfaceA #= Coupe*Hauteur,
```

```
  SurfaceB #= RestLargeur*Hauteur,
```

```
  partition(Pieces,ResultatA,ResultatB,SurfaceA,SurfaceB),
```

```
  labeling([Coupe]),
```

```
  coupe(ResultatA,Coupe,Hauteur,PlanA),
```

```
  coupe(ResultatB,RestLargeur,Hauteur,PlanB).
```

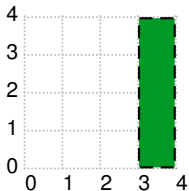
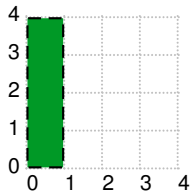
```
coupe(Pieces,Largeur,Hauteur,horizontal(Coupe,PlanA,PlanB)) :-
```

```
  ...
```

Éviter des symétries

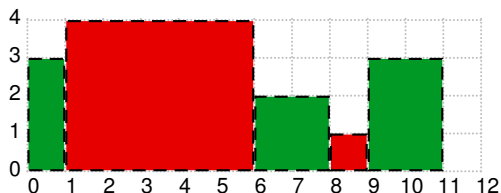
- On peut éviter la symétrie qui consiste en inverser la gauche et la droite, resp. le haut et le bas:
- Il suffit de renforcer la contrainte sur Coupe:

$$2 * \text{Coupe} \# = < \text{Largeur},$$



Une optimisation

- Idée : on peut avoir un cas de base plus général
- Nouveau cas de base : on essaye de placer tous les morceaux côte à côte (ou empilés).



Le programme complet (1)

```
% calcule la surface totale d'une liste de morceaux
sommessurfaces([],0).
sommessurfaces([(X,Y)|R],Somme) :- sommessurfaces(R,RSomme),
                                   Somme is (X*Y) + RSomme.

% ordonne les morceaux
ordonne([], []).
ordonne([(X,Y)|T],[(X,Y)|RT]) :- X=<Y, ordonne(T,RT).
ordonne([(X,Y)|T],[(Y,X)|RT]) :- X>Y, ordonne(T,RT).

coupe(Pieces,Largeur,Hauteur,Plan) :-
    sommessurfaces(Pieces,S),
    S =< Hauteur*Largeur,
    ordonne(Pieces,PiecesOnEnd),
    c(PiecesOnEnd,Largeur,Hauteur,Plan).
```

Le programme complet (2)

```
c(Pieces,Largeur,Hauteur,Plan) :-  
    listcoupe(Pieces,Largeur,Hauteur,Plan),!.  
  
c(Pieces,Largeur,Hauteur,vertical(Coupe,PlanA,PlanB)) :-  
    Coupe #:: [1..Largeur],  
    2*Coupe #=< Largeur,  
    RestLargeur #= Largeur-Coupe,  
    SurfaceA #= Coupe*Hauteur,  
    SurfaceB #= RestLargeur*Hauteur,  
    partition(Pieces,ResultatA,ResultatB,SurfaceA,SurfaceB),  
    nonempty(ResultatA), nonempty(ResultatB),  
    labeling([Coupe]),  
    c(ResultatA,Coupe,Hauteur,PlanA),  
    c(ResultatB,RestLargeur,Hauteur,PlanB).  
  
c(Pieces,Largeur,Hauteur,horizontal(Coupe,PlanA,PlanB)) :-  
    ...
```


Le programme complet (3)

```
listcoupe(Pieces, Largeur, Hauteur, list(Plan)) :-  
  Largeur #>= Hauteur, listcoupe_l(Pieces, Largeur, Hauteur, Plan).  
listcoupe(Pieces, Largeur, Hauteur, stack(Plan)) :-  
  Largeur #< Hauteur, listcoupe_l(Pieces, Hauteur, Largeur, Plan1),  
  renversepieces(Plan1,Plan).
```

```
renversepieces([], []).  
renversepieces([(X,Y)|L], [(Y,X)|L1]) :- renversepieces(L,L1).
```

```
listcoupe_l([],_,_, []).  
listcoupe_l([(X,Y)|R], Largeur, Hauteur, [(X,Y)|PlanR]) :-  
  Y #=< Hauteur,  
  RestLargeur #= Largeur - X, RestLargeur #>= 0,  
  listcoupe_l(R,RestLargeur,Hauteur,PlanR).  
listcoupe_l([(X,Y)|R], Largeur, Hauteur, [(Y,X)|PlanR]) :-  
  Y #> Hauteur,  
  X #=< Hauteur,  
  RestLargeur #= Largeur - Y, RestLargeur #>= 0,  
  listcoupe_l(R,RestLargeur,Hauteur,PlanR).
```

Est-ce qu'on peut faire encore mieux?

- Il y a des symétries quand on a des morceaux identiques
Exemple : `plan([(2,2),(2,2),(2,2)],4,4,Plan)`.
- Pour les éviter il faudrait changer la représentation du problème :
Lister des morceaux différents, avec un entier qui donne le nombre de copies souhaitées.
Exemple : `plan([(2,2):3],4,4,Plan)`.