Recherche Opérationnelle

Introduction à la Programmation Par Contraintes

ECL-3A

Alexander Saidi

2014-2015

I Une introduction pratique à la Programmation avec Contraintes

Propriétés de la CSP (*Constraint Satisfaction Problems*) et des techniques de programmation utilisant les contraintes (CP).

CP, CSP, CLP (et autres variantes et extensions comme COP, CFP, ...)

Remarque sur ce document :

• Le propos de ce chapitre est de donner un aperçu de la *Programmation avec Contraintes*.

• La plupart des exemples sont en BProlog,

• Quelques exemples permettent de compléter cet aperçu par des possibilités de résolution à l'aide des contraintes.

• Ce document reprend certains éléments abordés en introduction au chapitre précédent.

II LE PARADIGME DE SATISFACTION DE CONTRAINTES (CSP)

Contraindre = maintenir dans des limites

s'exercent sur les variables (paramètres)

Genèse : Affectation vs. Équation

- Test vs. Contrainte
- Ensemble de tests et d'affectations vs. un système d'équations / inéquations
- **Unification et différence** ($=$ *et dif*)

En C / C++ (impératif)	⇔	**En CSP**
if (X > 0) Y = X+1 ;	⇔	***X #> 0, Y #= X +1***
Si X>0 *Alors l'affectation a lieu...* *Exécution en séquence.*	⇒ ⇒	*On contraint X et Y* *On a (X=Y-1), propagations* *Véritable conjonction, équation / inéquation.*

➙ Statut et valeur initiale de X?

➙ notion d'attente et de retardement?

III UNE COMPARAISON AVEC LA PROGRAMMATION IMPÉRATIVE

Programmation **CLP/CSP**	Programmation **impérative**
Règle (et Ensemble de règles)	Procédure (et Programme)
Question / requête	Appel de procédure
Preuve	Exécution
Substitution, Unification, Matching	Passage de paramètres
Le programme = une recherche dans un espace combinatoire	Le programme implante un algorithme particulier
Aucun algorithme spécifique employé, seulement des heuristiques	Le contrôle est explicite (logique de Hoar/ Séquentielle)
L'exécution est "orientée-contraintes" (données)	L'exécution est "orienté-programme" (traitement)
Les contraintes maintiennent l'état de calcul	Les fonctions transforment un état de la mémoire vers un autre
Le modèle d'exécution est parallèle et concurrent	Le modèle de base d'exécution est séquentiel
Le dévérminage et l'optimisation sont plutôt globaux	Le dévérminage et l'optimisation sont locaux
Le raisonnement a lieu sur des réseaux de contraintes	

IV CSP : Une Idée intuitive

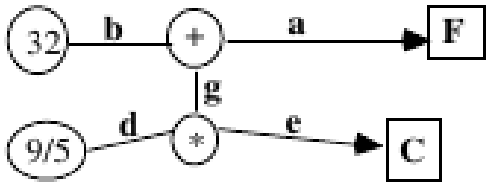
Pour un exemple simple comme la conversion F-C : $C = (F-32) * 5/9 \rightarrow F = 32 + 9/5 * C$

Les tous premiers systèmes (*Steele*) dans une vision différente de la programmation linéaire.

Les labels :

g = d*e ➙ e= g/d ➙d=g/e

a=b+g ➙b=a-g ➙ g=a-b



Règles de comportement des opérateurs (+, *, …) :

Ex : Si Z ← X+Y alors Y ← Z-X et X ← Z-Y

➙ Pour F = 212 :

- ➙ a = 212
- ➙ e = g / d = (a - b) / d
- ➙ e = (212-32) / d
- ➙ e = 180 / d = 180/9/5
- ➙ C =100

☞ Problème de cycle dans les graphes.

V Quelques Exemples introductifs

V-A Conversion C°/F°

- Définir une relation *linéaire* : ***F = M * C + B***
- 212 degrés F° = 100 degré C° : ***212 = M * 100 + B***
- 32 degrés F° = 0 degré C° : ***32 = M * 0 + B***

→ En Gprolog-RH (CLP-R) :

{F = M *C , 212 = M * 100 + B, 32 = M * 0 + B} *on obtient →* ***{ -1.8 * C + F = 0.0 } , B = 32.0, M = 1.8***

→ En **PIII** (gestion des nombres rationnels) :

{F = M *C , 212 = M * 100 + B, 32 = M * 0 + B}; *on obtient →* ***F = (9/5)C, M = 9/5, B = 32***

- De = à ≠ (PrologII = précurseur)
- Passages à ≥, ≤, … :

  ➥ Techniques : *Node/Arc/Path consistancy* (NC, AC, PC) + Back-Track.

  X in R et *Simplex*

- Domaines : Nombres : entiers, relatifs, réels (représentation par intervalles),

Bool, Rationnels, Symboliques, Chaines, Ensembles, ...

V-B Puzzle logique (simple)

TWO +

TWO

= *FOUR* *(Les lettres I= [T,W,O,F,U,R] sont des chiffres tous différents.)*

Principe de la solution :

- Soit L = [T,W,O,F,U,R] la liste des variables dont le **domaine** est 0..9
- On pose : tous les éléments de L deux à deux différents.
- On pose ensuite : **O+O=R+10*X1, X1+W+W=U+10*X2, X2+T+T=O+10*X3, X3=F**

  Ou **2*(100*T+10*W+O) = 1000*F+100*O+10*U+R**

- Finalement, si le système a plus d'une solution unique (proposé directement), on énumère des valeurs.
- Exemples de solution : {<1,3,2,0,6,4>}, {<7,3,4,1,6,8>}, etc.

V-C Une équation simple

```
nbr_tetes_pattes(Lapins, Pigeons,Tetes, Pattes) :-
  Tetes #= Lapins + Pigeons,
  Pattes #= 2 * Pigeons + 4 * Lapins.
```

- ?- nbr_tetes_pattes(Lapins, Pigeons, 6 , 10). ⇒ no.
- ?- nbr_tetes_pattes(Lapins, Pigeons,4 ,10). ⇒ Lapins = 1, Pigeons = 3
- ?- nbr_tetes_pattes(Lapins, Pigeons,Tetes, 10).

  ⇒ Lapins = 0..2, Pigeons =1..5 ,Tetes = 1..7

V-D Factorielle réversible

Factorielle (réversible avec des contraintes).

```
fact(0, 1).
fact(N, M) :- N #> 0 , N1 #= N-1 , fact(N1, M1) , M #= N * M1
```

Exemples de questions :

?- fact(3, X).⇒ X=6

?- fact(Y, 24), !. ⇒ X=4 % le cut ("! ") à voir plus loin.

V-E Amortissement (domaine des Réels)

Calcul de mensualité d'un prêt (réversible). Domaine des réels (ici, en Prolog-RH)

```
versement(Capital, Mois, Taux, Due, Mensualite) :-
   {Mois = 1, Due = Capital + (Capital *Taux - Mensualite)}.
versement(Capital, Mois, Taux, Due, Mensualite) :-
   {Mois >= 1, Mois1 = Mois -1 , Capital1 = Capital*(1 + Taux) - Mensualite ,
   versement(Capital1, Mois1, Taux, Due, Mensualite)}.
```

Des questions :

1) Pour 1000 Euros, remboursé en 1 mois, à 10% : combien doit-on payer à la fin du mois ?

➙ ***versement(1000, 1, 0.10, 0, M).*** ➙ ***M = 1100.0***

2) Pour 1000 Euros, remboursé en 12 mois, à 10% : quelle mensualités ?

➙ ***versement(1000, 12, 0. 10, 0, M).*** ➙ ***M = 146.7633151***

3) Remboursé en 12 mois, à 10%, 150 Euros par mois : quelle était la capital de départ ?

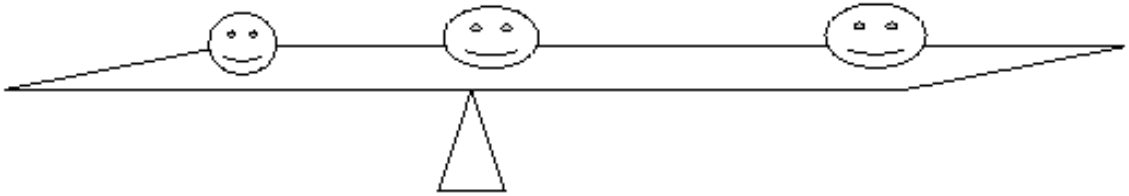
➙ ***versement(X, 12, 0.10, 0, 150).*** ➙ ***X = 1022.0537734***

➙ Voir un des exemples d'utilisation de **lp_solve** fournis avec BProlog.

V-F Balance

- 3 enfants (Jean, Pierre et Paul) sont assis sur une planche de 10 mètres (avec des repères sur l'intervalle **-5 .. 5**).
- Jean pèse 36 kg, Pierre 32 et Paul 16.

Comment s'assoir tel que la distance minimum entre chaque couple soit **supérieure à 2 mètres** et la planche s'équilibre ?



- Variables (emplacements X,Y,Z) et Domaines : **-5 .. 5**
- Contraintes (autres) : **X*36 +Y* 32 + Z * 16 = 0**

$$|X - Y| > 2,\ |X - Z| > 2,\ |Y - Z| > 2$$

- Solution : ➙ {-4, 2, 5},{-4, -4, 1},{-4, 5 -1} ...

N.B. : on accélère les choses si l'on impose $X < 0$ (casse les symétries)

V-G Pierres (exemple moins simple)

• Une pierre de 40 kg se brise en 4 morceaux. Avec ces morceaux, on peut peser tous les objets de 1 a 40 kg.

➙ Quel est le poids de ces morceaux?

```
pierre(L) :-
   L = [P1, P2, P3, P4],          % La liste des 4 pierres : P1, P2, P3, P4
   L : : 1..40,                   % Pi à valeur dans 1..40
   P1+P2+P3+P4 #= 40,             % Leur somme = 40
   P1 #=< P2, P2 #=< P3, P3 #=< P4,      % La relation entre les 4 pierres (pour accélérer)
   tous_les_kilos_pesables(1,40, L),     % Que l'on puisse peser de 1..40 kg avec les 4 pierres
   labeling(L).                   % énumérer des valeurs
```

• Tous les poids entiers entre A et B doivent être pesables.

```
tous_les_kilos_pesables(Piods_a_peser, Jusque, _On_s_en_fout_si_termine) :-
   Piods_a_peser = Jusque.      % On a tout testé, de 1 à 40 Kg.

tous_les_kilos_pesables(A_peser, Jusque, Liste_pierres) :-
   A_peser < Jusque ,                  % pas encore fini.
   peser(A_peser, Liste_pierres),      % peser 'A_peser' kg avec les 4 pierres
   Suivant is A_peser +1 ,
   tous_les_kilos_pesables(Suivant, Jusque, Liste_pierres).      % vérifier "Suivant"
```

• Chaque poids peut être d'un côté ou de l'autre de la balance (ou inutilisé pour certain poids)

➙ Envisager une disposition des 4 morceaux de pierres sur la balance de telle sorte que *A* kg soit "pesable " :

➙ Comment peser *A kg* à l'aide des 4 pierres P1,P2,P3,P4 ?

$P_i : i = 1..4, \quad B_j \in [-1,0,1], j = 1..4$

Une pierre P_i peut être

• sur le plateau gauche (B_i = -1 : sa valeur est soustraite du plateau droit),

• sur le plateau droite (B_i = +1 : sa valeur est ajoutée au plateau droit),

• absente (non utilisée) (c-à-d. B_i= 0 : sa valeur =0).

```
peser(A, [P1, P2, P3, P4]) :-      % Pour peser un poids donné A :
   L = [B1,B2,B3,B4],              % Choisir lesBitels que les 4 pierres disposés de part
   A #= B1*P1 + B2*P2 + B3*P3 + B4*P4,    % et d'autre de la balance fassent A kg.
   L : : [-1,0,1].      % Bi prend sa valeur dans -1 (à gche), 0 (absence) ou 1
```

• Test : ***pierre(L).*** $\Rightarrow$ ***L = [1,3,9,27].***

VI INGRÉDIENTS D'UN RAISONNEMENT EN CSP (DOMAINE FINI)

- **Pour résoudre un problème *P(X1, ..., Xn)* :**

1) Déclarer les domaines finis des variables du problème :

[X, Y, Z]∈1..10,

2) Exprimer les contraintes **2*X + 3*Y + 2 < Z,**

3) Rechercher des solutions (énumération si nécessaire, *indomain/1* ou *labeling/1*, *enumerer/1* ...) : **labeling([X]), labeling([Y]), labeling([Z]).**

?- [X,Y,Z] : : 1.. 10, 2*X + 3*Y + 2 #< Z, labeling([X,Y,Z]).

X = 1, Y=1, Z=8

X = 1, Y = 1, Z = 9

X = 1, Y = 1, Z = 10

X = 2, Y = 1, Z = 10

VII Propagation des contraintes : un exemple (dans N ou Z ?)

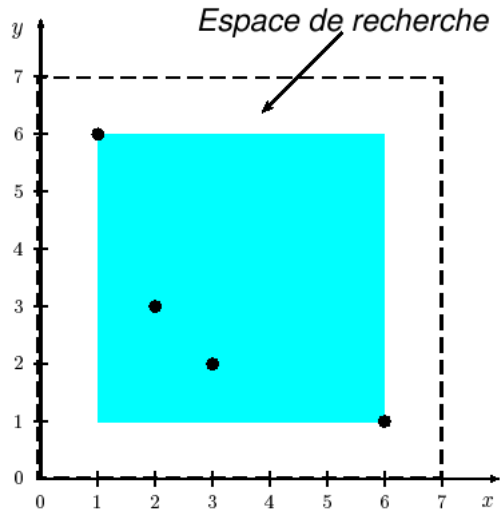
- Soient les contraintes du problème suivant (dans $\mathbb{N}$).

1) X in 0..7,
2) Y in 0..7,
3) X * Y = 6
4) X + Y = 5
5) X < Y.

I) La contrainte 3 permet d'éliminer 0 des domaines de X et de Y :

X in 1..6, Y in 1..6

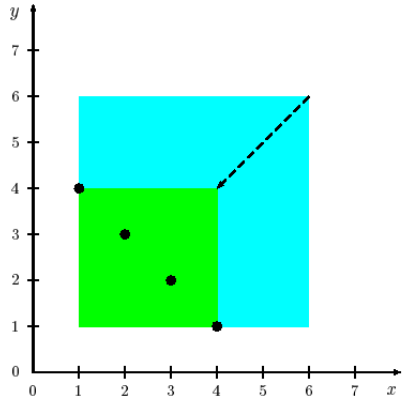
1) X in 0..7,
2) Y in 0..7,
3) X * Y = 6
4) X + Y = 5
5) X < Y.



II) La contrainte (4) permet une première réduction des domaines :

X in 1.. 4 , Y in 1.. 4

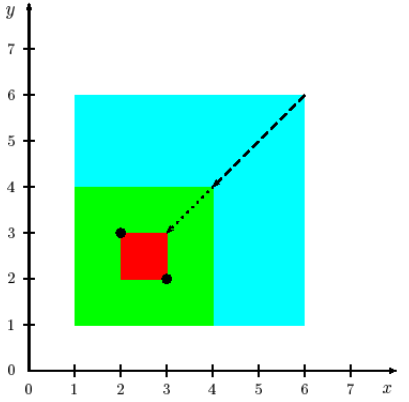
1) X in 0..7,
2) Y in 0..7,
3) X * Y = 6
4) X + Y = 5
5) X < Y.



III) Aussi, la contrainte (3) impose maintenant une seconde réduction (dans ℕ) :

X in 2.. 3 , Y in 2.. 3

1) X in 0..7,
2) Y in 0..7,
3) X * Y = 6
4) X + Y = 5
5) X < Y.



IV) Finalement, la contrainte (5) permet de choisir la solution **X=2, Y=3** (Zone hachurée)

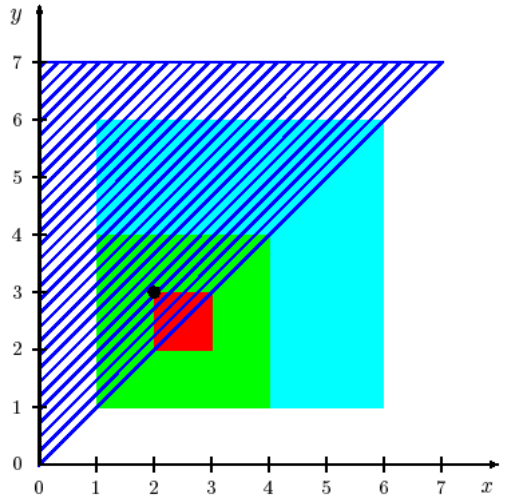
1) X in 0..7,

2) Y in 0..7,

3) X * Y = 6

4) X + Y = 5

5) X < Y.



N.B. : Une solution dans $\mathbb{Q}$: $\{X < 5 / 2,\ Y = 5 - X\}$

☞ Le CSP peut soumettre ce problème à un solveur Linéaire

Ou

• Procéder lui même par la propagation (si dans $\mathbb{N}, \mathbb{B}, \mathbb{Z}$).

☞ Voir aussi la notion de *Contrainte-Réponse*.

VIII ÉLÉMENTS DE PROGRAMMATION AVEC CONTRAINTES

- Traite des propriétés de la CSP et des techniques de programmation par contraintes (CP).
- Idée intuitive de contrainte : **Contraindre = maintenir dans des limites.**

VIII-A Objectifs de CSP

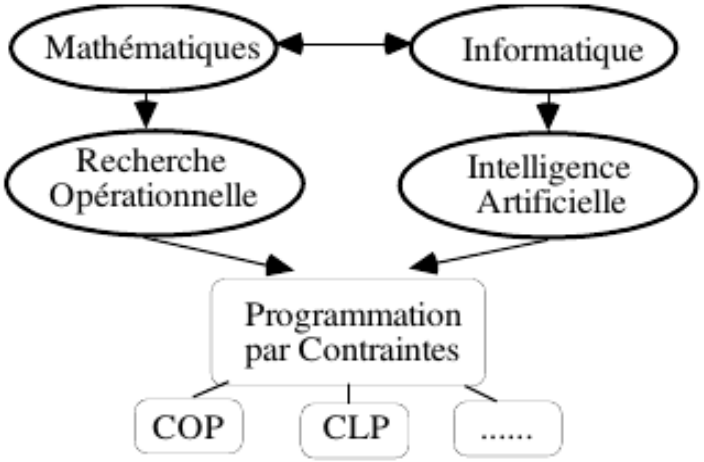
- La démarche de CP est nouvelle
- Les domaines d'applications
- Les bases (théoriques) de la CSP
- Les techniques de résolution des CSP
- Les bases théoriques de la CP
- Les méthodes et les techniques de la CP
- Formaliser un CSP donné : choisir la technique (la mieux adaptée) pour le résoudre

Appliquer les schémas de résolution :

- Identifier les variables et leur domaine
- Formaliser et exprimer les contraintes
- Ordonner, choisir et instancier les variables

- Vérifier les propriétés des solutions, Optimiser les solutions, ../..

VIII-B CSP multidisciplinaire



- CLP : Déclaratif, Relationnel, Fonctionnel (termes évalués).

  Analyse, définition et modification aisée

- Extension COP : Modélisation objet. Analyse, définition et modification aisée, Performances
- Extension CLPF : + fonctionnel

VIII-C Schéma CLP(X)

- CLP permet une *spécification déclarative* des problèmes ;
  la spécification séparée des techniques de résolution
- Le programmeur est libéré des *considérations opératoires* ;
  il se concentre sur la *définition des relations et les contraintes*
- Le système de programmation s'occupe de la *résolution dirigée par les contraintes*.
- Il n'y a plus de système de *résolution spécifique* à inventer pour chaque problème.
- Le programmeur formalise le problème dans le langage CLP en suivant les schémas de spécification.
- Les étapes de spécification d'une solution : absence des considération de codage

Faire un découpage hiérarchique et trouver une définition par contraintes ;
Spécifier les relations ;
Procéder éventuellement à une adaptation à l'outil ;
Spécifier les générations de valeurs ;
Vérifier (éventuellement) les propriétés des réponses ;
Optimiser éventuellement.

→ Maîtrise des méthodes, niveau requis

VIII-D Rappel du modèle utilisé (problème CSP)

V : Variables, **D :** Domaines, **C :** Contraintes, D $\subseteq$ C

Modèle : spécification des variables, leur domaines, leur sémantique déclarative et les contraintes conceptuelles qui les relient.

Contraintes effectives : Expression des propriétés de la solutions (et de la recherche).

Stratégie d'énumération : guidage de la recherche, choix des variables et de l'ordre sur (les valeurs) des domaines.

Plus précisement : étant donné $\mathbf{C(x_1, \ldots, x_n)}$ une contrainte sur des variables $\mathbf{V=\{x_1, \ldots, x_n\}}$, avec $\mathbf{x_1 \in D_1, \ldots, x_n \in D_n}$,

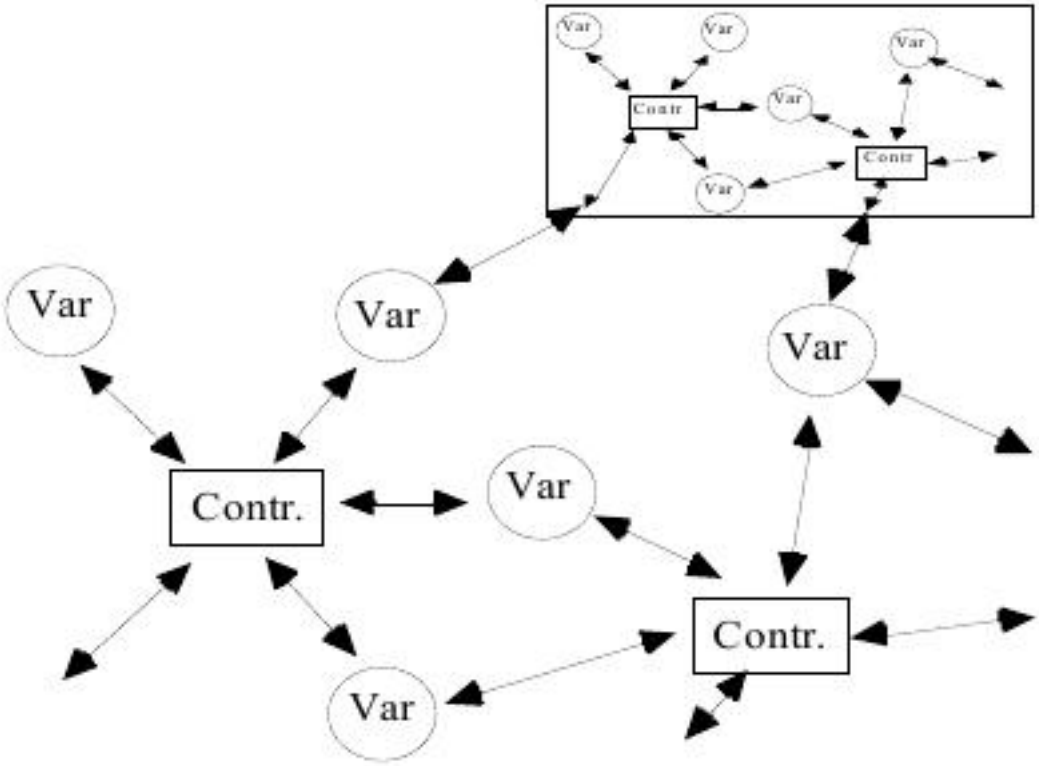
On appelle :

- **Espace de recherche** : le produit cartésien $\mathbf{D_1 \times \cdots \times D_n}$ des domaines des variables
- **Espace des solutions** : ensemble des n-uplets $\mathbf{(a_1, \ldots, a_n)}$ tels que $\mathbf{a_1 \in D_1, \ldots, a_n \in D_n}$ et $\mathbf{C(a_1, \ldots, a_n)}$ est vraie.

VIII-E Graphe des contraintes et Struct. d'un CP (V, D, C)

- Définition des variables (V)
- Expression des contraintes (D et C)
- Énumération éventuelle pour la recherche d'une/des solutions optimales.

Graphe des contraintes $\Rightarrow$



VIII-F Le rôle des contraintes dans un CSP

- Une contrainte représente une **relation** entre les objets
- Une contrainte peut spécifier :
  - Une information partielle / incomplète :

    l'age du capitaine est au moins 40 ans.
  - Une information floue :

    l'age du capitaine est environ 40 ans (intervalle)
- Une contrainte est déclarative : indépendante du processus de calcul
- Une contrainte n'est pas orientée, elle spécifie une relation :

  Dans $X + Y = Z$, si deux des 3 variables sont connues, la 3e est calculée.

  → L'ordre de l'expression des contraintes n'influence pas sur le sens du programme.
- En CSP, le domaine de calcul doit être connu :

Dans $X^2 = 2$

- Si le domaine est N → pas de solution
- Si le domaine est R → deux solutions

Dans $X^2 - Y^2 = 0$ & $X^2 + Y^2 = 2$.

le domaine de X et Y ne doivent pas changer dans la conjonction.

IX Aperçu de la complexité des problèmes Csp

Résolution de problèmes combinatoires (NP-Complet)

Exemples de combinatoire :

- Crypt-arithmétique (S,E,N,D,M,O,R,Y [E140?] {0..9})

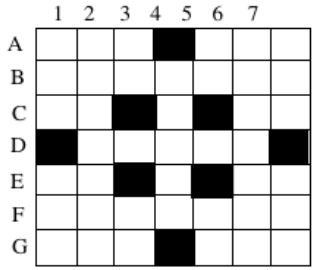
  S E N D + M O R E = M O N E Y

  e.g. 9 5 6 7 + 1 0 8 5 = 1 0 6 5 2

  10^8 combinaisons

- Composition de mots croisés (pour une grille ordinaire et un lexique de 150 mots équitablement répartis)

  3! 4! 11! 17! 15! 30! 33! 26! 5! solutions potentielles



Un exemple de grille 7 x 7

- Planification de la rotation de 10 bateaux dans 5 emplacements d'un port comporte env. $5^{10} \simeq 10^6$ poss. :

  ➙ **Trois minutes** de calcul pour un ordinateur testant une solution par milliseconde.

  Mais, pour 20 bateaux et 10 emplacements (10^{20})

  ➙ plus de **50 millions d'années** sur le même ordinateur !

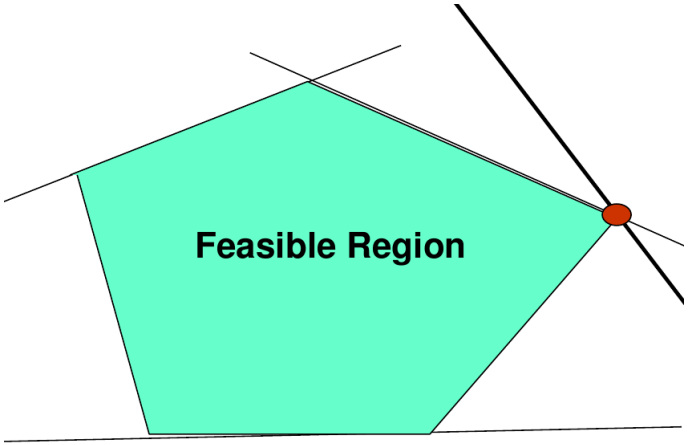
  Choix selon la complexité $|D|^{|V|}$

X Notion de Contrainte-réponse

Un programme transforme (simplifie, résout) les contraintes en entrées et produit les contraintes en sortie.

Les sortie sont appelées **contraintes-réponses**

Une contrainte-réponse est une évaluation partielle du programme : une instance plus spécifique du programme.



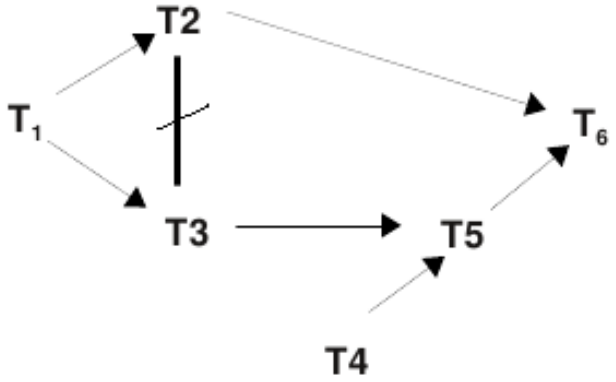
X-A Pert et contrainte-réponse dans N

Exemple de résolution de contraintes

- Par la technique (*Arc Consistency*)
- Variables (V) : $T_i, i = 1..6$ (des tâches)
- Le domaine (D) des tâches : {1, 2, 3, 4, 5}
- Les contraintes (Z) :

$T_1 < T_2,\ T_1 < T_3, T_2 \neq T_3, T_2 \neq T_3,\ T_2 < T_6,$

$T_3 < T_5,\ T_4 < T_5,\ T_5 < T_6$



$T_1 < T_2$

$T_1 < T_3$

$T_2 \neq T_3$

$T_2 < T_6$

$T_3 < T_5$

$T_4 < T_5$

$T_5 < T_6$

La propagation des contraintes :

- Si l'on propage les contraintes (directement puis par transitivité = *Arc-consistancy* puis *Path Consistancy*), les variables conservent les possibilités suivantes :

T1 = {1,2},T2 = {2,3,4},T3={2,3},T4={1,2,3},T5={3,4}T6={4,5}

- Si par exemple on pose **T1 = 2 :**

→ on obtiendra T1=2, T2=4, T3=3, T4={1,2,3}, T5=4, T6=5.

Consistance et simplification des contraintes de l'exemple (sous Bprolog) :

```
[T1,T2,T3,T4,T5,T6] : : 1..5,          % En Gprolog, écrire : fd_domain([T1,T2,T3,T4,T5,T6], 1,5)
T1 #< T2, T1 #< T3,T2 #\= T3, T2 #<T6, T3 #< T5, T4 #< T5, T5 #< T6.
```

T1 : : 1..2, T2 : : 2..4

T3 : : 2..3, T4 : : 1..3

T5 : : 3..4, T6 : : 4..5 Attention : ce ne sont pas les dates au plus tôt / tard

On pose T2 = 2 : cela déclenche d'autres simplifications

```
[T1,T2,T3,T4,T5,T6] : : 1..5, T1=2,
T1 #< T2, T1 #< T3,T2 #\= T3, T2 #<T6, T3 #< T5, T4 #< T5, T5 #< T6.
```

- ➙ T1 = 2
- ➙ T2 = 4
- ➙ T3 = 3
- ➙ T4 : : 1..3
- ➙ T5 = 4
- ➙ T6 = 5

XI NOTION DE HIÉRARCHISATION DES CONTRAINTES

Raisonnement hiérarchique

Un contrainte représente une propriété locale du problème à résoudre.

Les contraintes représentent les relations entre les différents objets du problème.

*resistance(V, I, R) :- V #= I * R.*

Les contraintes globales décrivent comment les contraintes locales interagissent.

Dans CLP, les propriétés **globales** sont représentées par les **règles**.

=> Les contraintes-réponses expriment des contraintes *globales*.

Le schéma de représentation des propriétés globales d'un CLP :

P(- - - -) :-

contraintes,

P1(- - - -), ..., Pn(- - - -)

XI-A Exemple indicatif

Exemple indicatif (en Gprolog-Rh pour placer des l'arithmétique sur les réels entre ") :

```
circuit_parallele (V, I, R1, R2) :-
        resistance (V, I1, R1)
        resistance (V, I2, R2)
        {I1 + I2 = I};
circuit_serie (V, I, R1, R2) : -
        resistance (V1, I1, R1)
        resistance (V2, I2, R2)
        {V1 + V2 = V}.
```

Expression d'une hiérarchie multi-niveaux :

```
circuit_parallele_serie(V, R1, R2, R3, R4) ->
        circuit_parallele (V1, I, R1, R2)
        circuit_parallele(V2, I, R3, R4)
        {V1 + V2 = V}.
```

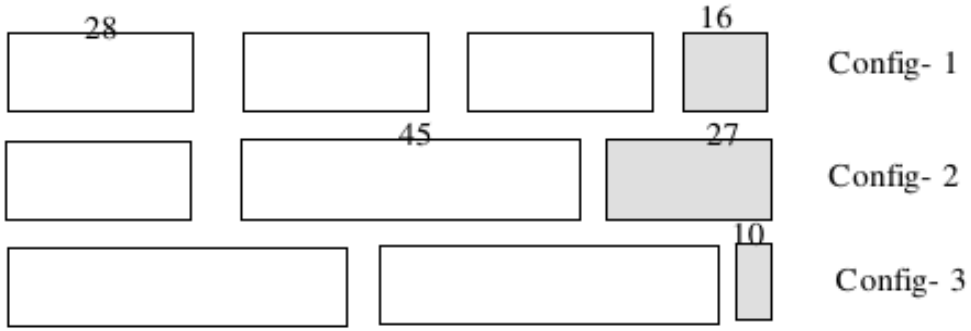
Ainsi, deux circuits parallèles sont montés en série (addition des V) auxquels est imposé la même valeur de I.

Ce dernière prédicat impose des contraintes aux deux autres qui à leur tour ont leur propres contraintes.

XII Aperçu de l'optimisation dans N : la découpe

Découpe de barres de 28 et 45 dans une barre d'un mètre.

Demande à satisfaire : 36 barres de 28 cm et 24 barres de 45 cm

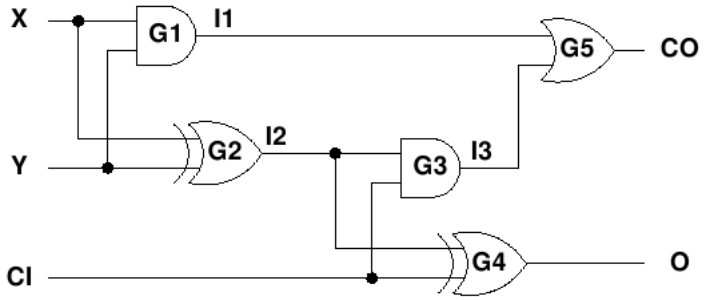



```
decoupe(Chutes, X,Y, Z) :-
    [X,Y, Z] : : 0 ..40,      % 0.. 36 suffirait (cas pire),
    , Chutes #= 16*X+27*Y+10*Z  % pas besoin de domaine pour  Chutes
    , 3*X + Y #= 36
    , Y + 2*Z #= 24
    , minof(labeling([ X,Y, Z]), Chutes)
    .
```

?- decoupe(C,X,Y,Z) . ➙ {C = 312, X = 12, Y = 0, Z = 12}

XIII Contraintes Booléennes

Exemple additionner



Modélisation :

Variables : entrées/sorties des portes

Domaines : [0,1]

Contraintes Booléennes (pseudo code) :

(I1 <=> X & Y), (I2 <=> X Oux Y), (I3 <=> I2 &CI) , % '&' : and, '|' : or

(O <=> I2 OuX CI), (CO <=> I1 | I3) % '<=>' : équivalence

Ou dans la notation canonique :

and(X, Y, I1), xor(X, Y, I2), and(I1, CI, I3),

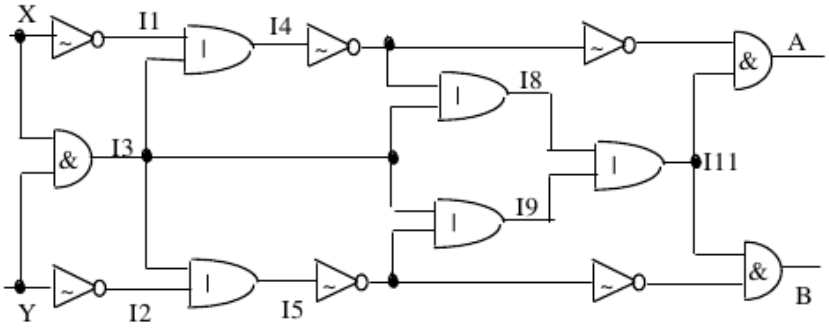
xor(I2, CI, O), or(I1, I3, CO), ...

XIII-A Simplification de Circuit électronique

- Simplifier le circuit suivant.
- Démontrer qu'il établit la relation A=Y, B=X.

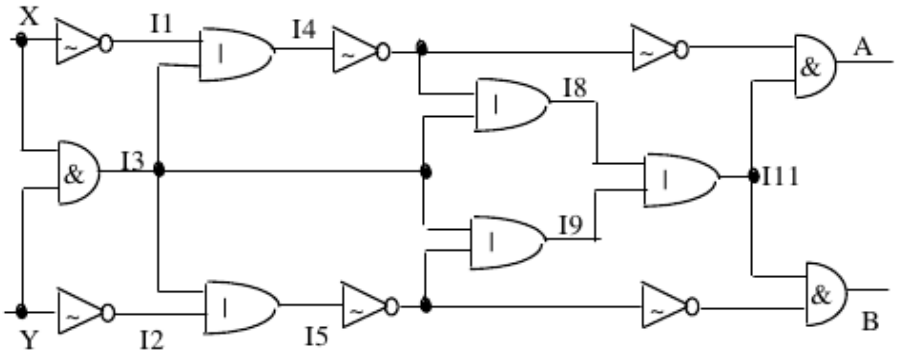
Rappel : contraintes booléennes :

#/\ : ET booléen
#\/ : OU booléen
#=> : implication,
#<=> : équivalence,
#\ : not
1 : true
0 : false



Le circuit est représenté par :

```
circuit(X, Y, A, B) :-
    I4 #<=> #\X #\/ I3,
    I3#<=>X #/\ Y,
    I5 #<=> #\Y #\/ I3,
    I8 #<=> #\I4 #\/ I3,
    I9 #<=> #\I5 #\/ I3,
    A #<=> I4 #/\ I11,
    I11#<=>I8 #\/ I9,
    B#<=>I5 #/\ I11.
```



Questions :

?- circuit(0,1,A,B) . ➙ {A = 1, B = 0}

?- circuit(1, 0, A, B) . ➙ {A = 0, B = 1}

☞ Certains systèmes donne la forme simplifiée symbolique (A=Y, B=X.)

Applications :

- Simplification, Optimisation, Simulation, ...

- Détection de pannes dans les circuits.

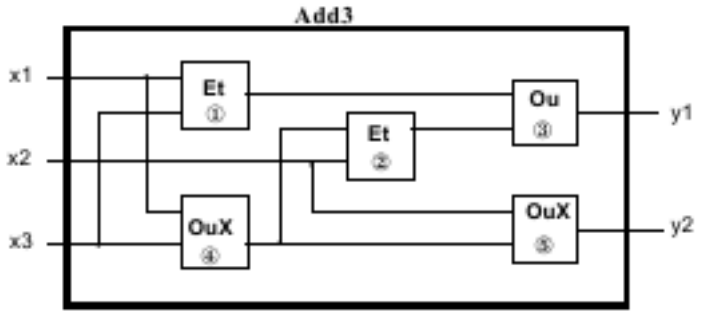
XIII-B Une application : exemple de détection de pannes

Circuit et pannes :

Soit un additionneur 3 bits :

On désire écrire un programme qui détecte une panne dans ce circuit selon l'hypothèse de panne unique : un seul composant est en panne à la fois.

Cette hypothèse est vérifiée par le prédicat *AuPlusUn-Vrai*.



Notons que ce prédicat permet d'installer une contrainte avant de décrire le circuit (méthode contraindre / générer).

Une valeur booléenne 1' sur les porte p1..p5 veut dire que celle-ci est en panne.

Une Contrainte comme **˜ p1 =>** (**u1** <=> **x1 & x3**) veut dire :

si P1 n'est pas en panne alors P1 vérifie la relation (électronique) qui lui est attribuée.

XIII-B-1 Une solution

Solution spécifique à ce circuit :

```
circuit([X1,X2,X3], [Y1,Y2], [P1,P2,P3,P4,P5]) :-
        #\P1 #=> ( U1 #<=> X1 #/\ X3 ),      % c-à-d : si P1 pas-en-panne, alors (implication) il fait son travail ...
        #\P2 #=> ( U2 #<=> X2 #/\ U3 ),      %          ... qui est d'établir l'équivalence entre U1 et (C1 & X3)
        #\P3 #=> ( Y1 #<=> U1 #\/ U2 ),
        #\P4 #=> ( U3 #<=> #\(X1 #<=> X3)),
        #\P5 #=> ( Y2 #<=> #\(X2 #<=> U3)),
    % Panne unique : au plus un VRAI dans la liste
      fd_at_most_one([P1,P2,P3,P4,P5]),
      fd_labeling([P1,P2,P3,P4,P5]).
```

Questions :

circuit([1,1,0],[0,1],[P1,P2,P3,P4,P5]).

% P1 = 0 P2 = 0 P3 = 0 **P4 = 1** P5 = 0 ? ; => Porte 4 en panne.

circuit([1,0,1],[0,0],[P1,P2,P3,P4,P5]). % P1 = 0, P2 = 0, **P3 = 1**, P4 = 0, P5 = 0 ? ; => P3 en panne

OU

% **P1 = 1,** P2 = 0, P3 = 0, P4 = 0, P5 = 0 => P1 en panne => une seule de P1 ou P3 est en panne

XIII-C Un autre exemple Booléen : Dieu !

Début de preuve d'existence de Dieu !

• 5 propositions de **George Boole.**

1- Quelque chose existe ;

2- Si quelque chose existe alors, soit quelque chose a toujours existé, soit les choses qui existent maintenant sont sorties du néant ;

3- Si quelque chose existe alors, soit ce quelque chose existe par la nécessité de sa propre nature, soit ce quelque chose existe par la volonté d'un autre être ;

4- Si quelque chose existe par la nécessité de sa propre nature alors quelque chose a toujours existé ;

5- Si quelque chose existe par la volonté d'un autre être alors

l'hypothèse que les choses qui existent maintenant sont sorties du néant, est fausse.

• L'ensemble de ces faits sera l'ensemble des arbres de la forme :

ValeurDeQuelqueChoseAToujoursExiste(X)

où X désigne la valeur de vérité pour la proposition "quelque chose a toujours existé".

Les proposition de George Boole sont codées par l'introduction de 5 variables qui sont les valeurs de vérité possibles des 5 propositions :

A : Quelque chose existe

B : Quelque chose a toujours existé

C : Tout ce qui existe maintenant est sorti du néant

D : Quelque chose existe par la nécessité de sa propre nature

E : Quelque chose existe par la volonté d'un autre être

Solution indicative : #\/ :OU, #/\ : ET, #\ : Négation, # => : Implication, #= : Egalité

```
valeurDeQuelqueChoseAToujoursExiste(B) :-
    A #= 1,
    A # => (B #\/ C) #/\ #\(B #/\ C),
    A # => (D#\/ E) #/\ #\(D #/\ E),
    D # => B,
    E # => #\ C .
```

Pour résoudre le problème, on pose la question :

ValeurDeQuelqueChoseAToujoursExiste(B). => **B=1**

Voir d'autres exemples / domaines en annexes.

XIII-D Booléens : Casse-tête logiques (Détente, Lewis caroll)

On considère les phrases d'un puzzle de **Lewis** Carroll. Il s'agit de répondre à des questions du genre :

"quel lien existe-t-il entre avoir l'esprit clair, être populaire et être apte à être député ?"

ou

"quel lien existe-t-il entre savoir garder un secret, être apte à être député et valoir son pesant d'or ?"

L'expression des propositions en Français :

1. Tout individu apte à être député et qui ne passe pas son temps à faire des discours, est un bienfaiteur du peuple.
2. Les gens à l'esprit clair, et qui s'expriment bien, ont reçu une éducation convenable.
3. Une femme qui est digne d'éloges est une femme qui sait garder un secret.
4. Les gens qui rendent des services au peuple, mais n'emploient pas leur influence à des fins méritoires, ne sont pas aptes à être députés.
5. Les gens qui valent leur pesant d'or et qui sont dignes d'éloges, sont toujours sans prétention.
6. Les bienfaiteurs du peuple qui emploient leur influence à des fins méritoires sont dignes d'éloges.
7. Les gens qui sont impopulaires et qui ne valent pas leur pesant d'or, ne savent pas garder un secret.
8. Les gens qui savent parler pendant des heures et des heures et qui sont aptes à être députés, sont dignes d'éloges.
9. Tout individu qui sait garder un secret et qui est sans prétention, est un bienfaiteur du peuple dont le souvenir restera impérissable.
10. Une femme qui rend des services au peuple est toujours populaire.

Le prédicat *CasPossible* suivant associe les phrases proposées aux variables booléennes sur lesquelles il pose les contraintes :

CasPossible(<
<a,"avoir l'esprit clair">, <b,"avoir reçu une bonne éducation">, <c,"discourir sans cesse">,
<d,"employer son influence a des fins méritoires">, <e,"être affiche dans les vitrines">,
<f,"être apte a être député">, <g,"être un bienfaiteur du peuple">, <h,"être digne d'éloges">,
<i,"être populaire">, <j,"être sans prétention">, <k,"être une femme">, <l,"laisser un souvenir impérissable">,
<m,"posséder une influence">, <n,"savoir garder un secret">, <o,"s'exprimer bien">, <p,"valoir son pesant d'or">>) ->
{ (f&˜ c) => g, (a&o) => b, (k&h) => n, (g&˜ d)=> ˜ f, (p&h) => j,
(g&d) => h, (˜ i&˜ p) => ˜ n, (c&f) => h, (n&j) => (g&l), (k&g) => i, (p&c&l) => e, (k&˜ a&˜ b) => ˜ f,
(n&˜ c) => ˜ i, (a&m&d) => g, (g&j) => ˜ e, (n&d) => p, (˜ o&˜ m) => ˜ k, (i&h) => (g|j) };

Le reste du programme vérifie qu'une liste fournie en entrée est un sous ensemble de la liste donnée dans *CasPossible*.

Possibilite(x) -> CasPossible(y) SousEnsemble(x,y);
SousEnsemble(<>,y) -> ;
SousEnsemble(<e>.x,y) -> ElementDe(e,y) SousEnsemble(x,y);
ElementDe(e,<e>.y) -> ;
ElementDe(e,<f>.y) -> ElementDe(e,y) {e#f};

Questions :

- Établissons les rapports éventuels qui existent entre "avoir l'esprit clair", "être populaire" et "savoir garder un secret" :

Possibilite(<p,"avoir l'esprit clair">, <q,"être populaire">, <r,"savoir garder un secret">>); => {}
La réponse est l'ensemble vide et signifie que selon Lewis Caroll il n'y a aucun lien entre ces trois vertus.

- Quel sont les liens qui unissent les propositions "savoir garder un secret", "être apte a être député" et "valoir son pesant d'or" ?

Possibilite(<p,"savoir garder un secret">, <q,"être apte a être député">, <r, "valoir son pesant d'or">>); => {p & q => r}
La réponse exprime le fait que si l'on sait garder un secret et que l'on est apte à être député alors on vaut son pesant d'or.

- Quel sont les liens entre les propositions "être une femme" et "être apte a être député" selon Lewis Caroll ?

Possibilite(<p,"être une femme">, <q,"être apte a être député">>); => {p & q = 0'}

Les deux sembles incompatibles ! ! gouja !

XIV Quelques types courants de contraintes

Voir à la fin de ce document pour le cas de Gnu-Prolog.

- Contraintes sur les domaines : ***B : : 2..25***

  [X,Y] in [1,3,5,7]
- Contraintes arithmétiques : ***A #= B, A #> B, A #\= B***
- Contraintes symboliques ***atmost(N, Liste, Val) element(Index, Liste, Val)***
- Contraintes booléennes : ***A #\/ B, A #=> B, A #⇔B***
- Contraintes d'évaluation ***#=(A, B, C : Bool) ou C #<=> (A=B)***

  (réification) ***#>=(A, B, C :Bool) #\/ (A, B, C :Bool)***
- Contraintes globales (niveau plus élevé de consistance)

  alldifferent([X1,...,Xn]) $X_i \neq X_j$, $i \neq j$

Exemple :

alldifferent([X1, X2, X3]) , [X1, X2, X3] : : 1 .. 2. → non.

Les incohérences/simplifications peuvent être détectées par avance.

- Contraintes définies par l'utilisateur : ../..

XV CONTRAINTES CONSTRUITES (PAR L'UTILISATEUR)

Dans un problème de tournée :

- *Un contrôleur visite des sites. Il visite le site* **S** *le jour* ***J****.*
- *Il ne visite chaque site qu'une fois ;*
- *Il peut visiter 2 sites en deux jours consécutifs ou espacés d'au plus 1 jour.*

Expression des contraintes : un exemple

• Une variable par site S_i , l'ensemble donne *Z=**Liste_des_sites***

• Domaine D de chaque élément de *Liste_des_sites* : les jours (1..7).

➙ ***Liste_des_sites in 1..7***

N.B. : certains environnements permettent des valeurs nominales (lundi, mardi, ...).

• Visiter chaque site une seule fois : ***alldifferent(Liste_des_sites)***

• Pour toute paire de sites S_1 et S_2 :

S1 #= S2 +X, I in 1..4, element(I, [-1,-2,1,2], X). voir aussi l'exemple

Dès que S1 prend une valeur, on élimine les valeurs incompatibles restantes pour S2.

N.B. on peut exprimer la même chose avec des disjonctions (coûteuses).

• Il suffit ensuite de générer des valeurs pour les éléments de *Liste_des_site.*

N.B. : divers environnements permettent des facilités de programmation

A titre d'indication, voici une solution (BProlog permettant de travailler dans Z)

```
visite(L) :-
   L : : 1..7,
   alldifferent(L),
   delais(L).
% les visites en jours consécutifs ou espacés d'un jour (autre solution).
delais([]).
delais([_X]) :- !.
delais([X,Y|L]) :-
      D : : [-2, -1, 1, 2],
      X #= Y + D,
      delais([Y|L]).
```

Appel : **length(L,5), visite(L), labeling(L).**

Solutions : L = [1, 2, 3, 4, 5], L = [1, 2, 3, 4, 6] ...

Voir aussi l'exemple PERT

N.B : labeling : énumération des valeurs dans un domaine.

XVI Un premier bilan

Les pages précédentes ont donné une introduction pratique de la CSP / CLP.
Les pages suivantes abordent ces aspects plus formellement et plus complètement :

- *Techniques de résolution et de satisfaction de contraintes Éléments de programmation avec des contraintes*
- *Complexité de ces problèmes*
- *CSP(X)*
- *Contraintes dans Bprolog*
- *Parcours de graphes de contraintes*
- *CLP : autres exemples*

XVII Techniques courantes de satisfaction de contraintes

XVIII Résolutions dans les environnements CSP

I) Traduction en contraintes plus basiques : 2*X + 3*Y + 2 < Z, est traduit en contraintes plus simples.

➞ à base de ***X in R*** **:** *étant donné les domaines de* ***X,Y et Z,*** *procéder à une réduction des domaine*

⇒ Cette réduction peut concerner seulement les bornes des domaines ou bien les valeurs des domaines :

```
'X=Y+C'(X,Y,C) :- X in min(Y)+C .. max(Y)+C, Y in min(X)-C .. max(X)-C.    % LookAhead partiel
```

➞ Exemple : si dom(Y)= {1,3,4,7,9} et C=2, alors X peut prendre une valeur dans 3..12 (extrémums)

```
'X=Y+C'(X,Y,C) :- X in dom(Y)+C ..Y in dom(X)-C.      % LookAhead complet
```

➞ Exemple : si dom(Y)= {1,3,4,7,9} et C=2, alors X peut prendre une valeur dans {3,5,6,9,11}

➞ les 3 tests de **consistances** + moteur **BT** (voir plus loin) ➞ ....

II) Résolution directe (sans simplification), par exemple à l'aide du Simplex particulier (cf. PIII et dif)

III) Résolution à base d'intervalles (Réels)

- Cas des booléen, réels, ensembles, ...

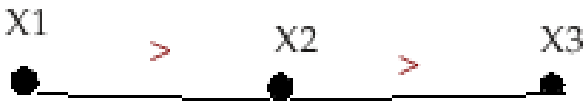
- Autres contraintes globales : au plus *K éléments d'une séquences doivent vérifier une relation R.*

XVIII-A X in R et Techniques de Consistance (Nœud, Arc et Chemin)

Il est possible de construire un solveur (complet) avec ces consistances + BT.

- Soit X = {X1,X2,X3}, D = {D1, D2, D3} = {1,2,3},

  Z = {X1 > X2 > X3}



- **Node Consistance** :

  ➙ Aucune simplification, les variables gardent leurs domaines.

  ➙ Les contraintes des domaines sont vérifiées.

  Exemple : **X in 1..3, X>3** n'est pas valide.

- **Arc Consistance :**

  ➙ Entre X1 et X2 ⇒ X1={2,3}, X2={1,2}. Puis indépendamment pour X2 et X3.

- **Path Consistance : (couteux)**

  ➙ On peut simplifier en : X1=3, X2=2, X3=1

XIX RETOUR ARRIÈRE

L'algorithme général de retour-arrière est rappelé ci-dessous :

```
Procédure Retour_arrière_chronologique(Z, D, C)
Début
  BT_1(Z, {}, D, C);
Fin Retour_arrière_chronologique;

Procédure BT_1(Libre, Label_composé, D, C) : Labels =
% Libre : ensemble de variables à étiqueter (Unlabelled)
% Label_composé : ensemble de labels déjà réalisés
Début
      Si (Libre={}) alors retourne (Label_composé)
      Sinon
       Prendre une variable X dans Libre
       Répéter
            Prendre une valeur V dans Dx
            Si (Label_composé + {<X,V>} ne viole pas C alors
             Résultat ← BT_1(Libre-{X},  Label_composé + {<X,V>},  D, C);
             Si (Résultat ≠ NIL) alors
                  retourne(Résultat);
             Fin si;
            Fin si;
       Jusqu'à (Dx = {});
       Retourne (NIL);
      Fin si;
Fin BT_1;
```

Remarques sur l'algorithme :

- Soit **n** : le nombre de variables,

  e : le nombre de contraintes et

  a la taille des domaines des variables du CSP.

  ➙ Il y a $\mathbf{a^n}$ combinaisons possibles de n-uplets (candidats : solutions potentielles).

- Pour chaque candidat, toutes les contraintes doivent être vérifiées une fois dans le cas pire.

  ➙ La complexité en temps de l'algorithme BT-1 est alors $\mathbf{O(a^n e)}$.

- Le stockage des domaines demande **O(na)** espaces.

  → Ce qui donne la complexité en espace de l'algorithme sachant que le stockage des labels_composés est de l'ordre de O(n) mémoire temporaire.

- La complexité en temps montre qu'une baisse de **a** permet un gain des performances.

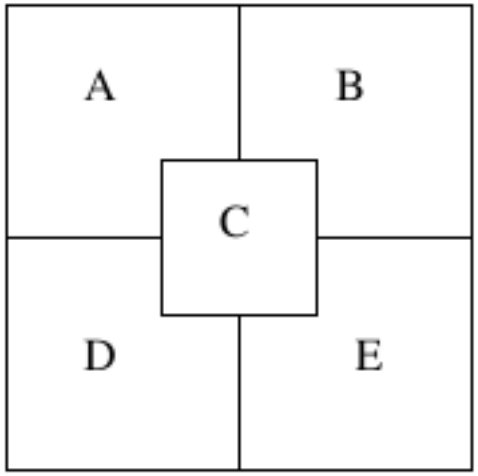
  → Cette réduction est effectuée par les techniques de réduction.

XIX-1 Exemple de coloration

Version générer tester + BT

⇒ On décide de se limiter à 3 couleurs [1,2,3]

```
coloriage_gen_test(A,B,C,D,E) :-
  % on énumère
  member(A,[1,2,3]),
  member(B,[1,2,3]),
  member(C,[1,2,3]),
  member(D,[1,2,3]),
  member(E,[1,2,3]),
  % on teste
  A=\=B, A=\=C, A=\=D,          % différences
  B=\=C,B=\=E,
  C=\=D,C=\=E,
  D=\=E.
```

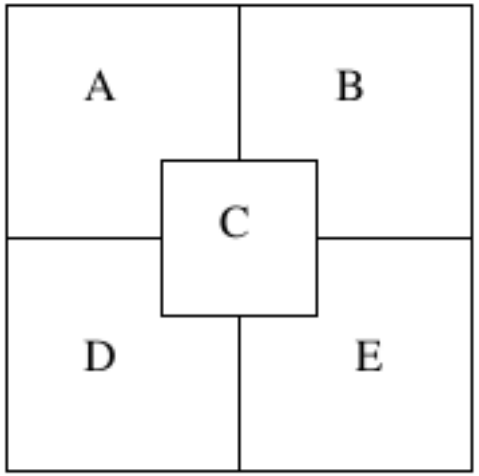


⇒ **Une solution** : A = 1, B = 2, C = 3, D = 2, E = 1

Version avec contraintes (Bprolog) :

- La différence (inéquation) se note "#=" (mais aussi **dif**).

```
coloriage_contr_gen(A,B,C,D,E) :-
    % On impose que les variables A,B,C,D,E valent 1, 2 ou 3
    [A,B,C,D,E] : : 1.. 3,
    % 2 régions voisines sont de couleurs différentes
    A#\=B, A#\=C, A#\=D,
    B#\=C,B#\=E,
    C#\=D,C#\=E,
    D#\=E,
    % énumèrer les valeurs des variables dans l'ordre A,B,C,D,E
    labeling([A,B,C,D,E]).
```



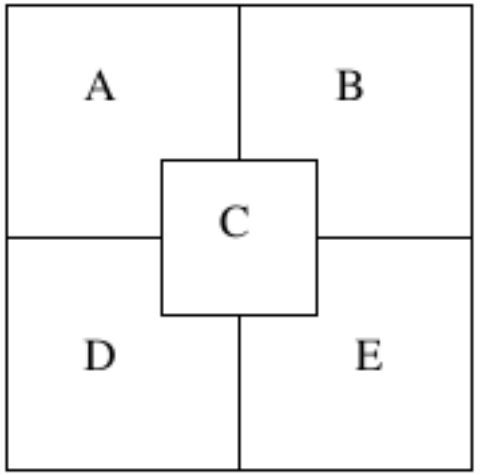
⇒ **Une solution** : A = 1, B = 2, C = 3, D = 2, E = 1

Version avec contraintes (Bprolog) :

- On introduit un ordre entre les variables pour éviter les solutions symétriques.

➙ On remplace #\= par #> pour éviter les solutions symétriques.

```
coloriage_contr_gen2(A,B,C,D,E) :-
    % On impose que les variables A,B,C,D,E valent 1, 2 ou 3
    [A,B,C,D,E] : : 1..5,          % 5 couleurs
    % 2 régions voisines sont de couleurs différentes
    A#>B, A#>C, A#>D,
    B#>C , B#>E,
    C#>D,C#>E,
    D#>E.
```



- Un exécution donne : A = 5, B = 4, C = 3, D = 2, E = 1

☞ Ce résultat est obtenu sans énumération.

- N.B. : le nombre de couleurs n'est pas optimum (à cause de #>) :

➙ C'est le 'prix' à payer pour éviter les solutions symétriques.

➙ Voir plus loin pour l'optimisation du nombre de couleurs.

☞ Les applications de la coloration : ...

Comment obtenir un nombre optimum de couleurs ?

- Une idée : sachant que les variables [A,B,C,D,E] prennent leurs valeurs dans [1..3], on a d'abord besoin d'un prédicat à base de contraintes permettant de calculer le maximum d'une telle liste (sans être obligé de travailler avec des valeurs).

➙ $max(Une\_liste)$ permet en BProlog d'obtenir le maximum d'une liste d'entiers (cf. BE2). Il peut être écrit par :

```
mon_max_liste([X],X).
mon_max_liste([X|R],M) :-
   M #= max(X,Y), mon_max_liste(R,Y).      % max(X,Y) représente le max des deux
```

- La seconde chose à savoir :

➙ pour la variable *V*, le prédicat $minof(But_V, V)$ permet d'obtenir une solution à *But* dans lequel la valeur de V est minimale (de même pour *maxof*).

- On peut maintenant formuler la question :

```
coloriage_contr_gen(A,B,C,D,E) ,
M #= max([A,B,C,D,E]),
minof(labeling(L), M).          % le lien entre la liste et M est fait ci-dessus
```

➙ Une réponse (les nums. couleurs sans trou) : [A,B,C,D,E] = [1,2,3,2,1], M = 3

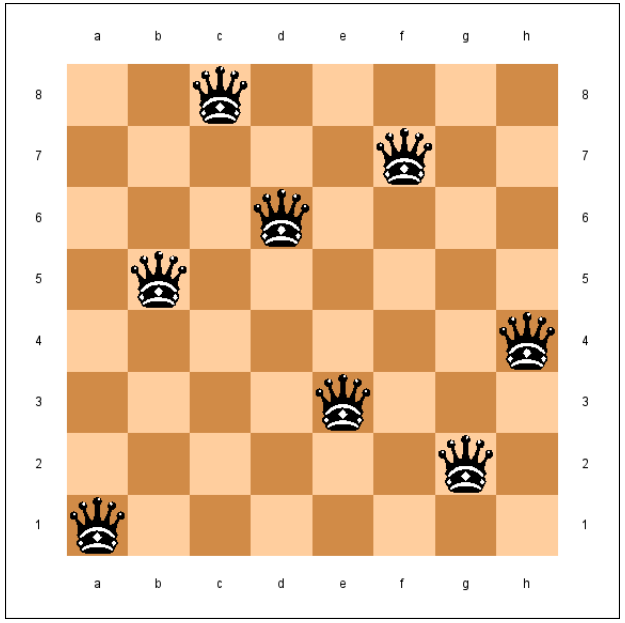
XX Modélisation et Complexité

- Le choix du modèle influe sur la complexité de la solution.

XX-A Illustration : Exemple N-reines

- Les contraintes à satisfaire : . . .

  une solution pour N=8 (→ la figure)



Choix de la représentation :

1- un ensemble S de 8 *variables* {X1..X8} chacune prenant une valeur dans l'intervalle D={A..H}. → 8^8

2- un ensemble S de 8 variables {A..H} chacune prenant une valeur dans l'intervalle D={1..8}. → 8^8

3- un ensemble S de 8 variables {V1..V8} chacune prenant une valeur dans l'intervalle D={1..64}. → $64^8 = 8^{16}$

4- un ensemble S de 64 variables {E1..E64} chacune prenant une valeur dans l'intervalle D={0,1}. → $2^{64} > 8^{21}$

...

Choix selon la complexité $|D|^{|V|}$

XX-B Formalisation de 8-reines

Identification des variables et leur domaines :

- Chaque ligne représente une variable : $Z=\{Q_1,...,Q_8\}$
- Le domaine de chaque variable est une colonne : $D_{Q1} = D_{Q2} = ... = D_{Q8} = \{1..8\}$

- Formalisation des contraintes :

1- Le fait de prendre une ligne comme variable assure que deux reines ne seront pas sur la même ligne : $C1 = \{\}$

2- Pour assurer que deux reines ne seront pas sur la même colonne :

$C_2 : \forall I,J : Q_I \neq Q_J,\ i,j \in \{1..8\}$

3- Pour assurer que deux reines ne seront pas sur la même diagonale :

$C3 : \forall I,J : si\ Q_I = a\ et\ Q_J = b\ alors\ I-J \neq a-b\ et\ I-J \neq b-a$

Exemple : $Q_1=1$, $Q_2 = 3$

Les domaines étant des entiers, on peut faire de l'arithmétique.

../..

Codage :

X/V_X représente la $X^{ème}$ ligne et la $V_X^{ème}$ colonne.

X variable, $V_X \in \{1..8\}$

compatible(X/V_X , Y/V_Y) réussit ssi (X/V_X, Y/V_Y) sont compatibles, étant données les contraintes du problème.

```
compatible(X/VX, Y/VY) :-
    VX #\= VY,          % C2
    X - Y #\= VX - VY ,     % C3
    X - Y #\= VY - VX .     % C3
```

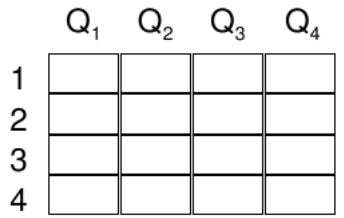
$Q_1, Q_2, Q_3, Q_4 \in \{1, 2, 3, 4\}$
$Q_1 \neq Q_2$, $Q_1 \neq Q_3$, $Q_1 \neq Q_4$,
$Q_2 \neq Q_3$, $Q_2 \neq Q_4$,
$Q_3 \neq Q_4$,

$Q_1 \neq Q_2-1$, $Q_1 \neq Q_2+1$, $Q_1 \neq Q_3-2$, $Q_1 \neq Q_3+2$,
$Q_1 \neq Q_4-3$, $Q_1 \neq Q_4+3$,
$Q_2 \neq Q_3-1$, $Q_2 \neq Q_3+1$, $Q_2 \neq Q_4-2$, $Q_2 \neq Q_4+2$,
$Q_3 \neq Q_4-1$, $Q_3 \neq Q_4+1$

- Pour N=8, il y a 8^8 combinaisons.
- **Formalisation des contraintes :**

- à titre indicatif (pour N=4) et par extension.

On peut généraliser pour tout N.



XX-C Une autre formalisation de N-reines

- $Z=\{Q_1,\ldots,Q_8\}$ pour représenter la position de chaque reine
- Le domaine de chaque variable est une case : $D_{Q1} = D_{Q2} = \ldots. = D_{Q8} = \{1..64\}$

Numérotation de gauche à droite, du haut vers le bas

- Pour une valeur N dans 1..64, la ligne et la colonne peuvent être calculées par :

ligne = (N / 8) +1 col = (N % 8) +1

- Étant donnés deux numéros de cases N1 et N2, compatible(N1,N2) réussit si les contraintes du problème (C1, C2 et C3) sont satisfaites :

```
compatible(N1,N2) :-
    Li1 #= (N1 / 8) +1, Col1 #= (N1 % 8) +1,
    Li2 #= (N2 / 8) +1, Col2 #= (N2 % 8) +1,
    Li1 #\= Li2, Col1 #\= Col2,      % C1, C2
    Li1 - Li2  #\= Col1 - Col2,      % C3
    Li1 - Li2  #\= Col2 - Col1.      % C3
```

- Pour N=8, il y a 64^8 combinaisons (labels composés) possibles !
- Cet exemple montre l'importance de la phase d'analyse d'un CSP.

N.B. : il existe une stratégie où chaque case est un booléen.

La formalisation booléenne est une combinatoire (2^{64}).

Remarques :

Le problème des N-reines est un **CSP binaire** :

- Chaque variable est contrainte par les autres variables (ça n'est pas toujours le cas dans les CSP)

- Pour tout N, l'attribution d'un label <l,c> à une variable la met en conflit avec seulement 3 autres.

=> Les autres variables auront N-3 possibilités.

=> Pour N=1000000, les autres variables auront 999,997 possibilités !

- Ces aspects (de ce problème particulier) ne sont présents que dans peu de CSP.

- Dans certaines versions, une fonction d'évaluation est introduite (pour des besoins d'optimisation)

=> par exemple, la distance totale entre toutes les cases de la solution (dispersion).

XXI Différents domaines CSP

Il est possible de travailler dans $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{Z}$, B, chaines, ensembles, ...

- L'exemple de conversion F-C en Eclipse donne (dans clpq) :

  F = M*C +B, 212 = M * 100 + B, 32 = B .

  → M = 9 / 5, B = 32

  % Linear constraints : {C = 5 / 9 * F}

XXI-A X dans CSP(X), autre que N, R, Z, Q

- Outre les contraintes linéaires dans N, R , Z, Q, B
- Les problèmes non linéaires sont peu pris en compte.
- Les autres X : Set (ensemble), String (monoïdes), Arbres, Temporelles, . . .
- Contraintes globales
- Contraintes définies par l'utilisateur

XXI-B Calcul en arithmétique non linéaire

• **N.B. :** Les systèmes actuels ne traitent pas les contraintes non linéaires.

Exemple : arithmétique non-linéaire

• Calculer la distance entre deux points parcourue à vitesse constante et sans accélération.

• On sait que si la vitesse est :

➙ diminuée de 20 Km/h, le temps du trajet augmente de 3 minutes ;

➙ augmentée de 20 Km/h, le temps du trajet diminue de 2 minutes ;

Formulation en Prolog-RH :

{D=V*T, D=(V-20)*(T+3), D=(V+20)*(T-2)}.

➥ On obtient une simplification

{ - (V * T) + D = 0.0 },

{ D - V * 3.0 - T * V + T * 20.0 = -60.0 },

{ D + V * 2.0 - T * V - T * 20.0 = -40.0 }

Un autre exemple : multiplication des nombres complexes (non linéaire)

zmul(R1, I1, R2, I2, R3, I3) :-

R3 = R1 * R2 - I1 * I2

, I3 = R1 * I2 + R2 * I1.

➥ On obtient sous Prolog-RH :

{ - (I1 * R2) - I2 * R1 + I3 = 0.0 }

{ I1 * I2 - R1 * R2 + R3 = 0.0 }

Puis un question :

?- zmul(1, 2, R2, I2, R3, I3).

➥ On obtient :

{ -0.5 * I3 + 2.5 * I2 + R3 = 0.0 }

{ -0.5 * I3 + 0.5 * I2 + R2 = 0.0 }

- La plupart des systèmes CLP *retardent* l'évaluation des contraintes non linéaires.

XXI-C Domaines ensemblistes

Manipulation des ensembles (opérateur intersection /\ )

```
?- Car : : {renault} .. {renault, bmw, mercedes, peugeot}
     , Type_french : : {renault, peugeot, citroen}
     , Choice #= Car /\ Type_french .
```

➙ ***Choice = {renault} .. {renault, peugeot}.***

Autres Exemples (Bprolog) :

- V : : {}..{a,b,c} : V est un sous ensemble de {a,b,c} incluant l'ensemble vide.
- V : : {1}..{1..3} : V est un des ensembles {1},{1,2}, {1,3}, et {1,2,3}.

  L'ensemble {2,3} n'est pas une valeur possible pour V.
- V : : {1}..{2,3} : échec car {1} n'est as un sous ensemble de {2,3}.

Bprolog propose des prédicats de conversion entre Liste ↔ ensemble

XXII Autres Contraintes

XXII-A Contraintes sur les chaînes

Exemple : ***Ch1 : 3, X = Ch1 . L . Ch1, X : 10*** . *% taille de Ch1=3*

XXII-B Contraintes globales

atmost(2, [X1 , X2 , X3 , X4 , X5], 1)

au plus deux variables parmi X1..X5 sont égales à 1

alldifferent([X1 , X2 , X3 , X4 , X5])

les variables {X1 , X2 , X3 , X4 , X5 } sont 2 à 2 différentes

minimisation / maximisation, Contraintes disjonctives (PERT) : ...

XXII-C Contraintes sur les Arbres / Graphes

Arbre comme structure de tout terme

- ➙ Notion d'arbre infini (X=f(X))
- ➙ Contraintes définies par l'utilisateur

  => Symboliques

  => Temporelles

(voir plus loin pour un exemple sur les graphes).

XXII-D Retour à l'expression de contraintes par l'utilisateur

Nous avons vu un exemple plus haut.

En BProlog, à l'aide des clauses avec garde, l'utilisateur peut spécifier différents types de contraintes.

Aussi, on peut dire qu'un prédicat comportant des contraintes est en soit une extension des contraintes.

Par exemple, on peut trouver le maximum d'une liste de nombres dont on connait pas les valeurs.

```
max_contrainte([X], X).
max_contrainte([X | L], M) :-
  M #= max(X,M1),  % le prédéfini  max/2
  max_contrainte(L, M1).
```

Questions :

?- max_contrainte([A, B, C, D], 3). % imposer : A, B, C, D =< 3.

?- max_contrainte([A, B, C, D], M), A=2, B #= A+1, C #> B, D #= A + B - C .

A = 2, B = 3, C = 4..5, D = 0..1, M = 4..5

Ce qui donne (après énumération) :

A = 2, B = 3, C = 4, D = 1, M = 4

A = 2, B = 3, C = 5, D = 0 , M = 5

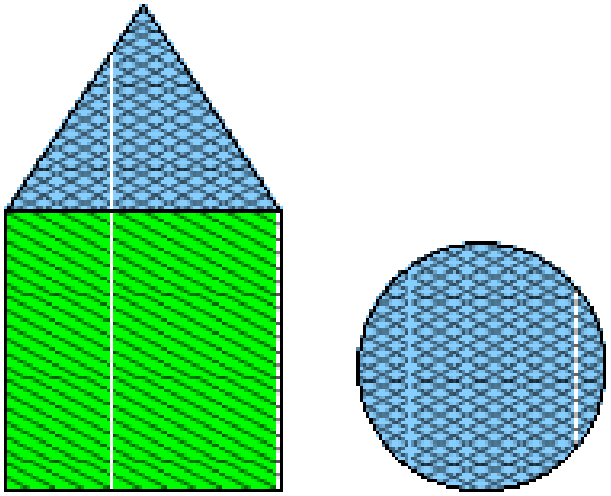
XXII-E Contraintes symboliques

- Les contraintes sont exprimées sur des arbres (comme pour l'unification Prolog).

  Par exemple : 2 arbres (termes composés) seront identiques si leurs composants le sont.

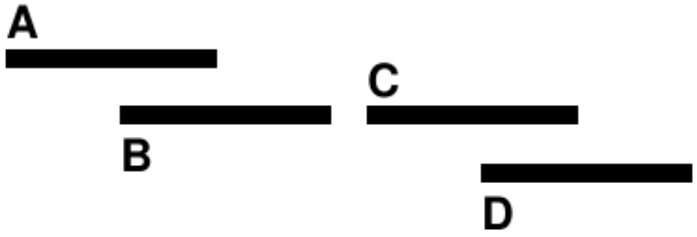
  ➙ Ou 2 arbres sont différents (selon le principe de l'algorithme d'unification)

- On peut manipuler des contraintes symboliques :



bleu(X) & sur(X,Y)
➙ X = triangle
Y = rectangle

XXIII Contraintes temporelles (réalisables par l'utilisateur)



Par exemple, de :

rel(AoverlapB, BbeforeC, R_AC)

& *rel(BbeforeC,CoverlapD, R_BD)*

& *rel(R_AC, CoverlapD, R_AD)*

On obtient :

R_AC= AbeforeC

& *R_BD=BbeforeD*

& *R_AD=AbeforeD*

Voir plus loin pour les détails des contraintes temporelles.

XXIII-A Aperçu du raisonnement Temporel

Un **évènement** :

- est un intervalle de temps dont la durée est nulle. ;
- correspond à une "pointe" de temps.

Un intervalle de temps :

- est associé à un processus qui a une durée ;
- exprime 2 évènements **début** et **fin** d'un processus (fini)

- Le raisonnement temporelle exploite les informations contenues dans les ensembles d'intervalles de temps et d'évènements qui expriment une relation d'ordre.

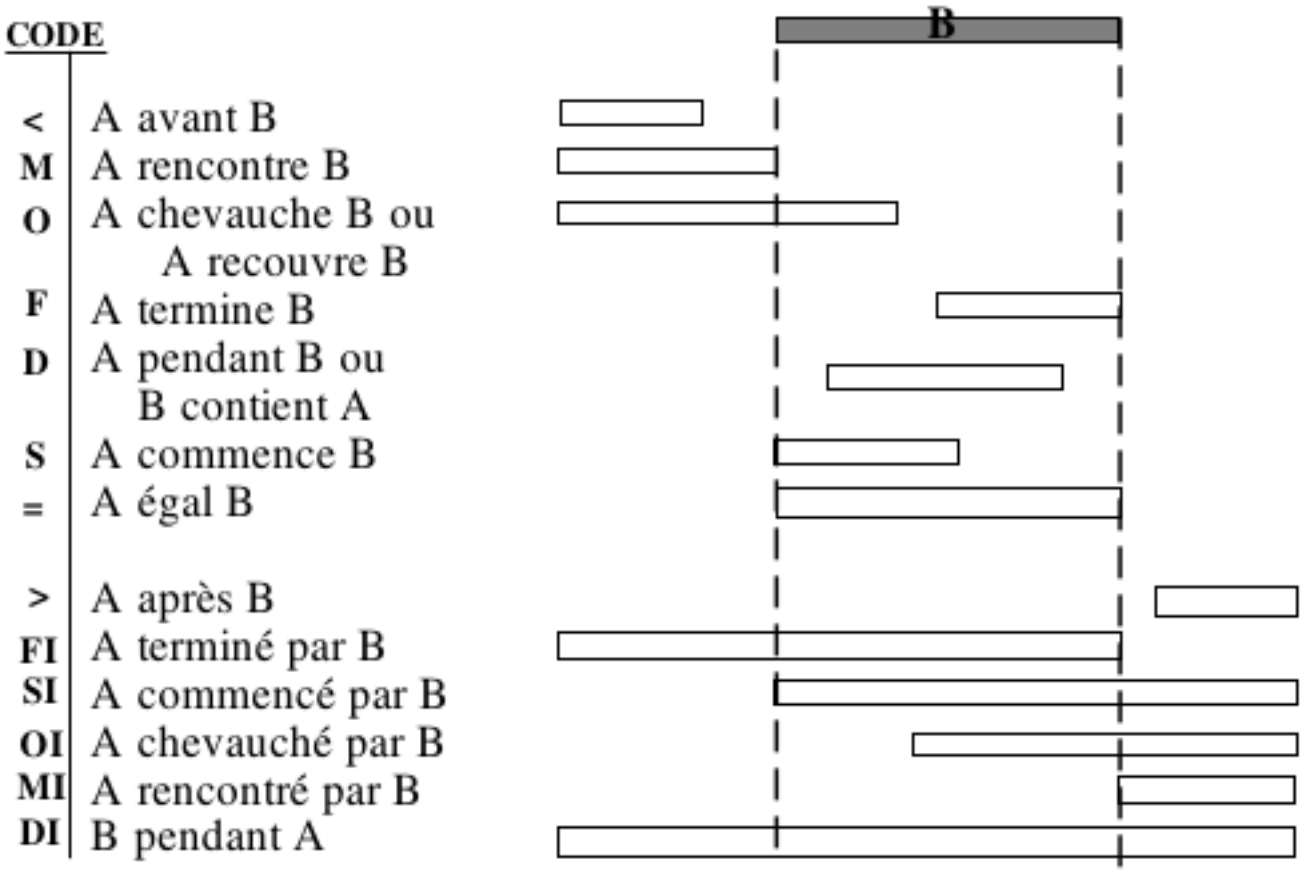
  Il permet d'inférer de l'information à propos des relations entre les intervalles de temps.

- Si l'on considère uniquement des évènements (durée nulle) alors 3 relations suffisent :

  A avant B, *B avant A* et *A égal B*.

XXIII-B L'algèbre d'intervalles (du temps)

Relations temporelles primitives



13 relations temporelles primitives (Allen)
Raisonnement sur les durées des événements

Formulation avec les contraintes

- Chaque variable représente une relation temporelle entre 2 évènements.
- Pour n évènements, il y a **n(n-1)/2** relations (variables).
- Le domaine des variables est l'ensemble des 13 primitives temporelles.

Les contraintes (propriétés) du temps :

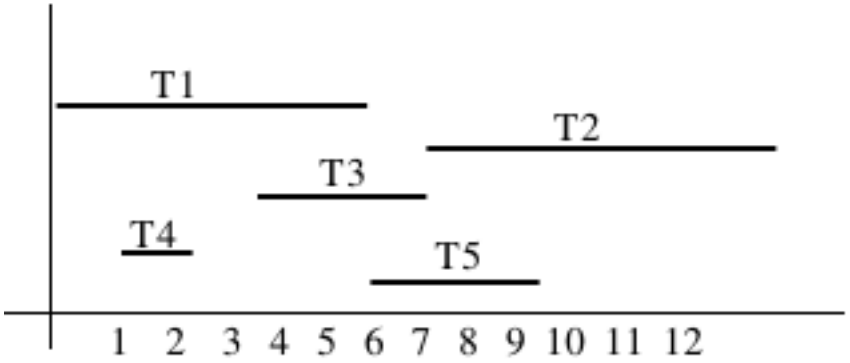
- Si A avant B et B avant C alors A avant C.
- Si A chevauche (overlap) B et B chevauche C alors A doit

  chevaucher , rencontrer (commence tout de suite après l'autre) ou être avant C.
- Si A termine B et B rencontre C alors A rencontre C
- Si A termine B et B commence par C alors A pendant C
- Si A rencontre B et B chevauche C alors A avant C.
- ……

XXIII-C Exemple (ordonnancement)

Soit un ensemble de tâches T1 .. T5 avec des relations temporelles et leur durées :

- *T1 avant T2*
- *T3 rencontre T2*
- *T4 pendant T1*



- *T5 chevauche T2*

Durées (T1..T5) : 5, 6, 3, 1, 3

But : On veut résoudre le système temporel ci-dessus en trouvant le début et la fin de chaque tâche.

Une solution →

avant(process(N1,D1, L1), process(N2,D2, L2)) :- D1+L1 < D2, D1 >=0.

% A avant B /\ B avant C => A avant C.

avant(process(N1,D1, L1),process(N2,D2, L2)) :-

process(T3), avant(process(N1,D1, L1),T3),

avant(T3,process(N2,D2, L2)).

rencontre(process(N1,D1, L1),process(N2,D2, L2)) :- D1+L1 = D2, D1 >=0.

rencontre(A,C) :-

process(B), chevauche(A,B), chevauche(B,C).

% A termine B /\ B rencontre C => A rencontre C.

rencontre(A,C) :-

process(B), termine(A,B), chevauche(B,C).

pendant(process(N1,D1, L1),process(N2,D2, L2)) :-

D1 > D2, D1+L1 < D2+L2, D2>=0.

% A termine B /\ B commencé par C => A pendant C.

pendant(A,C) :-

process(B), termine(A,B), commence_par(B,C). ....

XXIV Les contraintes dans Bprolog

Ces lignes sont tirées de la documentation (html et pdf) du Bprolog disponible dans le même répertoire que BProlog.

Contraintes sur les arbres :

freeze(X,Goal).

frozen(Var). ce qui est gelé.

dif(T1, T2).

Contraintes Domaine fini

Vars in Domaine ou *Vars : : Domaine.* ou *domain(Vars, Inf, Sup).*

Vars notin Domaine.

fd_var(V) : est-ce une variable de domaine ?

fd_new_var(V). créer une nouvelle variable de domaine

fd_min/fd_max/fd_min_max(V). le min/max/les deux d'une variable de domaine.

fd_size(V,N) la taille du domaine

fd_dom(V,L) L est la liste des éléments du domaine de V

fd_true(V,E) E est un élément du domaine de V

fd_set_false(V,E) exclure E du domaine de V

➙ Un exemple : X : :1..10, fd_var(X), fd_dom(X,L). ➙ X = _0960 : :[1..10] ➙ L = [1,2,3,4,5,6,7,8,9,10]

fd_next(V,E,Next) Next est la prochaine valeur après E du domaine de V.

fd_prev(V,E,Prev)

fd_include(V1,V2) le domaine de V1 inclue celui de V2 au sens ensembliste

fd_disjoint(V1,V2) le domaine de V1 et celui de V2 sont disjoints

fd_degre(V,N) le nbr de contraintes reliées à V est N

fd_vector_min_max(Min, Max) fixer la taille (min/max) du vecteur de bits représentant le domaine d'une variable (dont certaines valeurs sont exclues, sinon, le domaine est un intervalle).

Contraintes arithmétiques (#=)/2 - constraint equal,

(#\=)/2 - constraint not equal,

(#<)/2 - constraint less than,

(#=<)/2 - constraint less than or equal,

(#>)/2 - constraint greater than,

(#>=)/2 - constraint greater than or equal

Contraintes Globales

Énumérations

Optimisations

Contraintes booléennes

(#\)/1 - NOT,

(#<=>)/2 - equivalent,

(#\)/2 - XOR,

(#=>)/2 - imply,

(#/\)/2 - AND,

(#\/)/2 - OR,

#\/2 : différents

>, >=, <, =< au sens habituels mais sur les booléens

count(Val, Liste, Rop, N) : N éléments de la Liste respectent la relation Rop avec Val (i.e. E_i *Rop Val*, $E_i \in Liste$)

alldifferent/1

...

Contraintes sur les ensembles

Interface vers lp_solve

XXV Parcours de graphes (de contraintes) dans les CSP

Un solveur parfait donnerait toutes les solutions sans énumération.

En l'absence d'un solveur parfait, il faut mettre en place des heuristiques .

Les heuristiques sont associées aux procédures de recherche.

Des techniques heuristiques intéressantes en CSP :

- **First-Fail** Pour réussir, essayer d'abord où vous risquez plus d'échouer.
- **Forward-Check / Look ahead** Anticiper le future pour réussir le présent
- **B & B**

Néanmoins, l'ensemble de ces techniques s'appuient sur l'algorithme général de propagation.

XXVI Techniques Branch & Bound (séparation-évaluation)

La technique générale **Branch-and-Bound** consiste à diviser le problème en plusieurs sous-problèmes (phase de séparation permettant de découper un coût global en sous coûts)

Puis, dans la phase d'évaluation,

à trouver une évaluation de la borne du problème (par exemple le coût de la solution minimale/maximale du problème dans le cas de minimisation / maximisation).

Cette approche évite l'énumération de tout l'espace de recherche :

aussi tôt que l'on a trouvé une solution avec un coût de référence (de quelque manière que ce soit), on ne prend plus en considération les problèmes dont l'évaluation de la borne est moins bonne que celle de la solution trouvée (la référence).

Si une autre solution avec un coût moindre se présente, celle-ci deviendra la référence, etc.

Le coût de référence constitue une contrainte globale sur l'ensemble des arbres de recherche développés.

Exemple :

On suppose avoir trouvé une borne maximale de **450 km** pour la distance *Paris-Lyon* (celle correspondant aux traits doubles).

On commence à reprendre l'itinéraire passant par *Pontoise* (considéré comme le sous- problème "aller de Paris à Lyon en passant par Pontoise").

Les autres sous-problèmes seraient "aller de Paris à Lyon en passant par Auxerre", ...

Il est simple de constater qu'ayant parcouru *Paris* - - *Pontoise* - - *Reims* sans atteindre la destination, on peut arrêter la recherche dans cette branche pour examiner éventuellement un tout autre itinéraire.

En Bprolog, **minof** et **maxof** permettent d'avoir accès à cette heuristique. Par contre, la réalisation de cette optimisation peut être faite de différentes manières (simplement sur les domaines, à la Simplex, etc.)

XXVI-A Efficacité de l'heuristique B&B

La méthode B&B peut reprendre une démonstration déjà faite.

On reprend l'exemple précédent (ou fd_simple reprend les preuves).

p(3). p(2). p(1).

q(6). q(5). q(7).

r(12). r(11). r(10).

fd_simple(T) :- T #= A+B+C, p(A), q(B), r(C). *% différentes valeurs ... T=16*

% B & B manuelle en minimisation

manual_bb(T) :- fd_simple(Ref), suite(T,Ref).

suite(T, Ref) :-

Sol #< Ref, fd_simple(Sol), *% la même solution est reproduite sous contraintes de cout*

!, suite(T,Sol).

suite(T, T).

Une version plus efficace évite de reprendre les preuves depuis le début.

bb_efficace(_T) :- save_cout(9999), fd_simple(Ref), maj_cout(Ref), fail.

bb_efficace(T) :- lire_cout(T).

save_cout(Val) :- global_set(cout, Val).

lire_cout(Val) :- global_get(cout, Val).

maj_cout(Val) :- global_get(cout, Old_Val), (Val #< Old_Val -> save_cout(Val); true).

Question

?- bb_efficace(T). ➙ T = 16

XXVII Le principe First Fail

- Décider d'un ordre dans les tests pour faire d'abord celui qui risque davantage d'échouer :

choix de variable dont le domaine est le plus réduit

- Décider d'un ordre dans les générations des valeurs (énumération) pour affecter une valeur aux variables les plus contraintes.

- Il existe plusieurs stratégies d'énumération :

- La-plus-contrainte d'abord (en nbr de contraintes)

- La-plus-contrainte d'abord (en taille du domaine)

XXVIII LA TECHNIQUE FORWARD-CHECK

Dès qu'une variable prend une valeur, on installe des contraintes permettant d'éliminer les valeurs (devenues incompatibles) des domaines des autres variables.

En GProlog, ces techniques sont à la fois proposées par le système et également utilisées par le programmeur.

Exemple : différents tests de N_reines dans la version contraindre-générer (classique)

Rappel de la solution classique (sans heuristique) :

```
queens(N, L) :-
 fd_domain(L,1,N),
    safe(L),
    fd_labeling(L).
safe([]).        % plus rien à contraindre
safe([X|L]) :-      % Les contraintes d'un pion (X)  par rapport aux autres
     noattack(L,X,1), % contraindre X à respecter les autres (en commençant par 1)
     safe(L).
```

Vérification des contraintes par le prédicat safe/1 :

Deux pions ne doivent pas être sur la

même ligne (C1)

même colonne (C2)

même diagonale (C3).

=> On prend N variables, une par ligne => C1 satisfaite.

=> Pour les autres contraintes :

```
noattack([],_,_).
noattack([Y|L], X, I) :- % I progresse de 1 à N (pour couvrir toutes les colonnes)
        diff(X,Y,I),     % contraintes entre les deux pions X et Y
        I1 is I+1,
        noattack(L,X,I1). % Contraintes entre X et les autres
diff(X,Y,I) :-       % Ici, on vérifie effectivement les contraintes entre deux pions
        X#\=Y, X#\=Y+I, X+I#\=Y.
```

XXVIII-A Simulation Forward-Check

On installe le domaine puis on énumère une valeur.

On élimine ensuite certaines valeurs des domaines des autres variables par NOATTACK.

% Version Forward Check de N-reines

queens_fc(N, L) :- length(L,N), fd_domain(L, 1, N), forward_check(L).

%Traiter un pion (par rapport aux autres), puis réitérer avec les autres.

forward_check([]) .

forward_check([X|L]) :- fd_labeling(X), noattack(L, X, 1) , forward_check(L).

noattack([],_,_).

noattack([Y|L],X,I) :- % I progresse de 1 à N (pour couvrir toutes les colonnes)

diff(X,Y,I), % contraintes entre les deux pions X et Y

I1 is I+1,

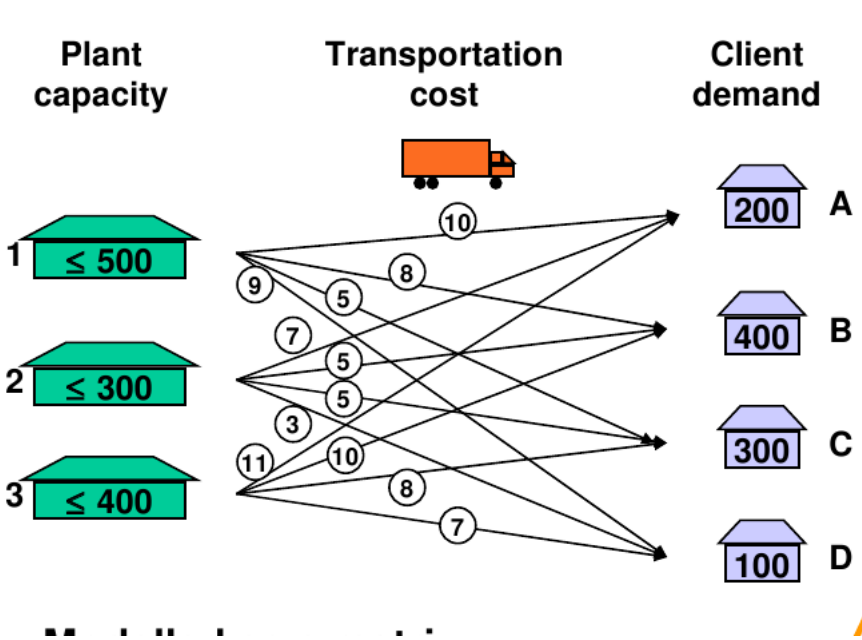
noattack(L,X,I1). % Contraintes entre X et les autres

diff(X,Y,I) :- % Ici, on vérifie effectivement les contraintes entre deux pions

X#\=Y, X#\=Y+I, X+I#\=Y.

XXIX CLP PAR DES EXEMPLES

XXIX-A Un exemple en Logistique




```
minimize
        10 A1 + 7 A2 + 11 A3 +
         8 B1 + 5 B2 + 10 B3 +
         5 C1 + 5 C2 +  8 C3 +
         9 D1 + 3 D2 +  7 D3

subject to
        A1 + A2 + A3 = 200
        B1 + B2 + B3 = 400
        C1 + C2 + C3 = 300
        D1 + D2 + D3 = 100

        A1 + B1 + C1 + D1 ≤ 500
        A2 + B2 + C2 + D2 ≤ 300
        A3 + B3 + C3 + D3 ≤ 400
```

Spécification (en programmation Mathématique) ***Spécification via une Matrice***

N.B. : pour la programmation dans R, voir Gprolog-RH (ou BProlog, Eclipse ...)

A1	A2	A3	B1	B2	B3	C1	C2	C3	D1	D2	D3		
1	1	1	0	0	0	0	0	0	0	0	0	=	200
0	0	0	1	1	1	0	0	0	0	0	0	=	400
0	0	0	0	0	0	1	1	1	0	0	0	=	300
0	0	0	0	0	0	0	0	0	1	1	1	=	100
1	0	0	1	0	0	1	0	0	1	0	0	≤	500
0	1	0	0	1	0	0	1	0	0	1	0	≤	300
0	0	1	0	0	1	0	0	1	0	0	1	≤	400
10	7	11	8	5	10	5	5	8	9	3	7		

XXIX-B Car sequencing

• Chaîne de montage de voitures comportant des sections spécialisées selon les options :

- Toit ouvrant
- Radiocassette
- Air conditionné
- Traitement antirouille
- ABS

• Quand une voiture passe dans une section, les ouvriers la suivent et effectuent le travail. Ils ont le temps nécessaire pour installer l'option.

• On ne peut pas installer une chaîne lente pour permettre d'installer toute option. Il faut concevoir la chaîne de telle manière qu'elle puisse traiter un pourcentage déterminé de voiture pour chaque option.

Un cas concret :

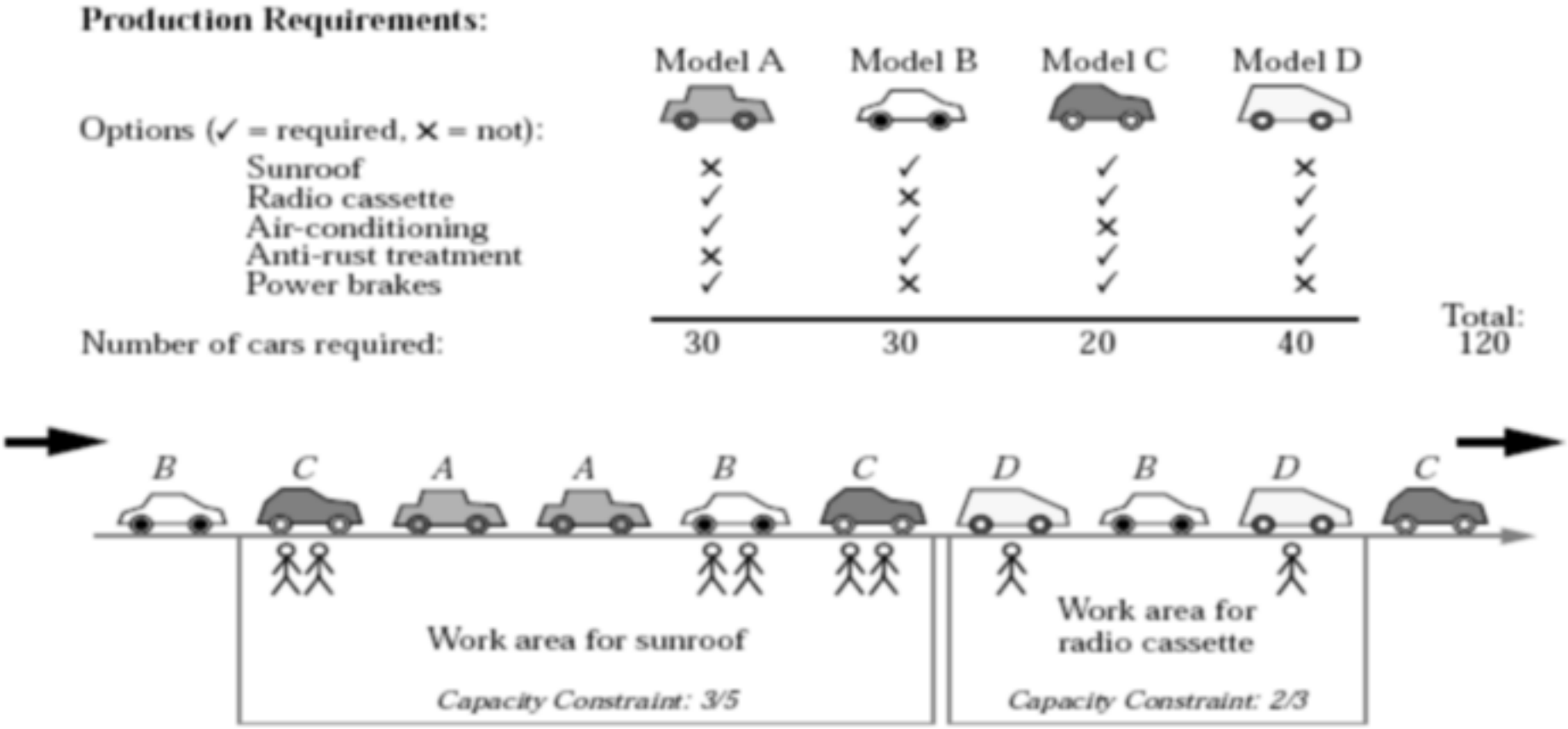
- 60% des voitures demandent de "toit ouvrant".
- 5 voitures passeront devant la section "toit ouvrant" pendant l'installation sur une voiture.

➙ Pour pouvoir installer le toit ouvrant sur 60% des (5) voitures, il faut donc 3 équipes d'installation sur ce poste.

- La chaîne à une capacité de 3/5 pour l'option "toit ouvrant".
- Lorsqu'une voiture passe devant ce poste, une équipe disponible commence à s'en occuper en suivant la voiture jusqu'à l'extrémité de la section.
- Pendant ce temps d'installation, 5 autres voitures seront passées devant ce poste.
- Si 2 d'entre elles nécessitent le toit ouvrant, les deux autres équipes pourront les équiper.
- Toutefois, si une 4ème voiture nécessite cette même option, le poste ne sera pas assez rapide pour réagir à temps.

Les données :

- 120 voitures à équiper (par jour).
- 4 modèle de voitures {A, B, C, D}.
- Pour chaque modèle, les options possibles sont connues.
- On connaît également les capacités des postes.



Le but :

Ordonner les voitures sur la chaîne de telle sorte que toutes les capacités soient satisfaites.

Remarques :

L'utilisation des démonstrateurs de théorèmes et des systèmes experts n'ont pas abouti à la résolution du problème.

Complexité de l'ordre de $4^{120} = 1000^{24}$

Problème résolu de façon efficace par les techniques CSP (environnement CLP).

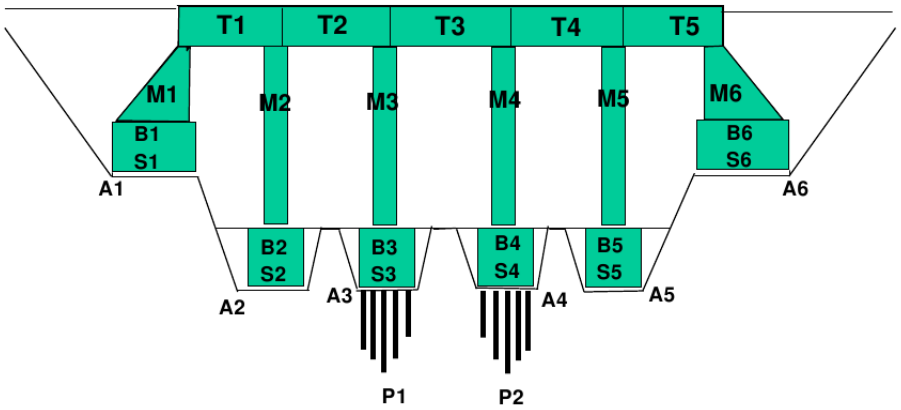
Démarche :

- Une *variable* pour représenter la voiture sur la chaîne de montage (120 variables ici, N en général).
- Le *domaine* de chaque variable est l'ensemble des modèles {A, B, C, D}.
- Le but est d'assigner une valeur à chaque variable (une position dans la chaîne de montage) satisfaisant les contraintes de capacité et les exigences de la production.

La solution est présenté sur le schéma :

.... B C A A B C D B D C ....

XXIX-C Construction de pont



No.	Name	Description	Duration	Resource
1	PA	Beginning of project	0	-
2	A1	Excavation (abutment 1)	4	excavator
3	A2	Excavation (pillar 1)	2	excavator
...				
8	P1	Foundation Piles 1	20	pile-driver
9	P2	Foundation Piles 2	13	pile-driver
10	U1	Erection of temporary housing	10	-
11	S1	Formwork (abutment 1)	8	carpentry
...				
17	B1	Concrete Foundation (abut. 1)	1	concrete-mixer
...				
23	AB1	Concrete Setting Time (abut 1)	1	-
...				
29	M1	Masonry work (abutment 1)	16	bricklaying
...				
35	L	Delivery of preformed Bearers	2	crane
36	T1	Positioning (preformed bearer 1)	12	crane
...				
42	V1	Filling 1	15	caterpillar
...				
46	PE	End of Project	0	-

Les contraintes temporelles :

- Précédence :

before(Start1,Duration1,Start2)

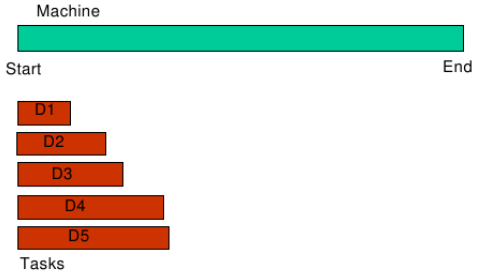
- Delay Maximal

before(Start2, -Delay, Start1).

Avec **before(Start1,Duration1,Start2) :-**

(Start1 + Duration1 =< Start2).

Les contraintes de ressources :



noclash(S1, D1,S2,D2) :- (S1>=S2+D2).

noclash(S1, D1,S2,D2) :- (S2>=S1+D1).

XXX Annexe A : Quelques Exemples d'application

Conception de matériel informatique

- vérification de circuits
- connexion des couches de circuits
- moins efficace que le code dédié mais plus flexible
- utilisateurs : **Dassault, Siemens**

Placement d'objets

- placement de containers
- remplissage de containers
- utilisateurs : **Michelin**

Problèmes de d´coupage

- minimisation de pertes lors de la d´coupe de matériaux (papier, verre, bois, métaux, ...)
- utilisateur : **Dassault**pour les pièces d'avion
- performances dépendent du contexte
- papier : facile, programmation lin´aire
- métaux : plus difficile, programmation par contraintes

Allocation d'espace

- portes pour les avions
- quais pour les trains et les bateaux
- utilisateur : **Aéroport CDG**

Allocation de fréquences

- trouver des fréquences radio pour les :
- téléphones portables
- communications radio
- armée, ...

Ordonnancement de la production

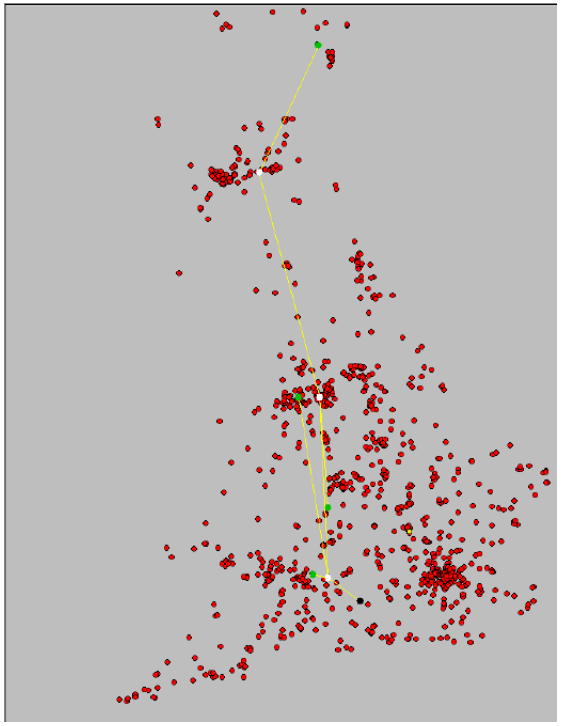
- planifier les tâches sur des machines dans une usine
- le plus important succès de la PPC
- meilleures performances que la RO
- plusieurs installations commerciales
- librairies dédiées à l'ordonnancement (**ILOG** scheduler)

Emploi du temps

- emploi du temps
- construction d'horaires de personnel :
- santé, commerce, usine, ...
- planification des équipages sur les avions, les trains
- tournée de véhicules
- ...

Des exemples récents en Australie (Monash University)

Logistique avec Dépôts (Australie) :



Objectifs :

Génération de plannings (tableaux de bord)

Minimiser conducteur/véhicule/couts de transport

Contraintes :

Disposer de fenêtre pour les ramassages, chargements et livraison

Capacité des véhicules

Données sur le temps de voyage

Données sur le temps de chargement et déchargement

Données sur le " shift " du personnel

Retombées attendues :

Stratégie : quel client servir (contrat ?)

Profit : combien et comment charger (tel trajet n'apporte pas / beaucoup / . . .)

Flight Schedule Retimer :

Objectifs :

(Re) planification des vols

Respect des contraintes

Minimiser les changements aux planifications existantes

Bénéfices attendus :

Gains en utilisation de la flotte

Changements de profile des quais (slots)

Amélioration de la ponctualité

Network Resilience (élasticité, adaptabilité de réseaux)

Objectifs :

Pour chaque " route " passant par un nœud N donné, trouver une route qui ne passe pas par N.

Evaluer le coût et les risques

Assurer la bande passante sur les routes alternatives

Retombés :

Diminuer les redondances (matériels, nœuds)

Maintenir la qualité de service (QoS)

Exploitations des technologies diverses

Autres :

- Répartition de Charges et Routage (Internet Routing and Balancing)
- Logistique (Supply Chain)
- Transport
- Planification de construction (Construction Scheduling)
- Systèmes de santé (planification, optimisation des ressources, ...)

Caractéristiques de ces problèmes :

Plus de 10,000 variables, plus de 50,000 contraintes (y compris disjonctives), Méta-heuristiques et B&B

XXXI Autres exemples d'applications de CSP

XXXI-A Planning & Scheduling

cf. l'exemple *car sequencing.*

Voir aussi les Tps.

XXXI-B Traitement des langues naturelles (TLN)

- L'analyse syntaxique est un cas typique de CSP.
- En TLN, chaque mot a un nombre fini de rôles.
- Le langage restreint le domaine de ce rôle qu'un mot peut remplir (e.g. nom, verbe, adverbe, ...).
- La grammaire d'un langage restreint les rôles qu'une séquence de mots peut jouer simultanément.
- Un des objectifs de l'analyse est d'identifier le rôle de chaque mot.
- C'est cet objectif qui peut être considéré comme un CSP.

XXXI-C Bases de données (BD)

- L'instanciation des variables dans une requête est un CSP.
- Dans le domaine d'optimisation des requêtes dans les BD, les techniques CSP peuvent être appliquées.
- Inversement, les techniques développées en recherche en BD peuvent servir en CSP.

Les techniques d'optimisation des requêtes en BD dont le but est d'améliorer les performances des moteurs de recherches (cloisonnement de l'espace de données, les techniques de parcours d'arbres, ...) sont des exemples de ce transfert de technologies entre les deux technologies.

XXXI-D Isomorphisme de graphes

Dans les réseaux sémantiques, on peut chercher la présence d'un concept (représenté sous forme de graphe).

Graphe matching (Graph morphism)

Étant donné 2 graphes G1 et G2, on veut vérifier si G2 contient un sous graphe qui correspond à G1.

Le graphe (V1,E1) contient le graphe (V2,E2) si :

- Tout noeud de V2 correspond à un nœud distinct dans V1 ;
- Pour les noeuds x1,y1 dans V1 et x2,y2 dans V2 tels que x2,y2 correspond à x1,y1, si (x2,y2) est une arête de E2 alors (x1,y1) est une arête de E1.

Exemple et la formalisation CSP pour montrer que G2 contient G1 :

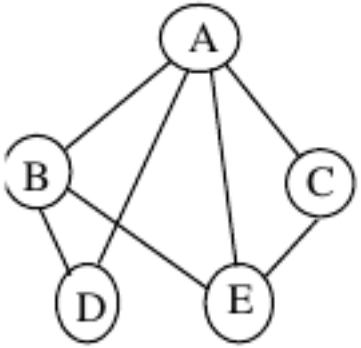
• Les variables : {A,B,C,D,E},

• Les domaines {a,b,...,i}.

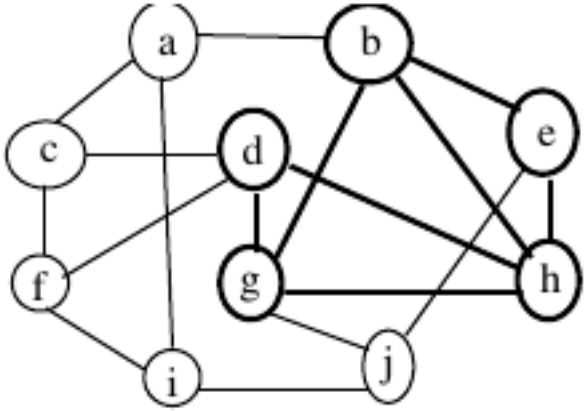
• Les contraintes : pour tout label composé (<x,p><y,q>), si x et y sont connectés dans G1 alors p et q le sont dans G2.

Par exemple, (<A, h><B, g>) satisfait la contrainte sur A et B car (g,h) est une arête de G2.

• Une solution : le label (<A,h><B,g><C,e><D,d><E,b>) .



graphe G1



graphe G2

TABLE DES MATIÈRES