Ecole Centrale de Lyon
2A

Algorithmes & Raisonnement (AR)

Introduction à la Programmation Par Contraintes (PPC ou CP)

Alexandre Saidi
M. I . 2008-2009

Table des Matières

I- INTRODUCTION À LA PROGRAMMATION AVEC CONTRAINTES 6
I.A- Le paradigme de Satisfaction de Contraintes (CSP)....7
I.B- Une Idée intuitive (exemple de conversion F-C)....8
I.C- Exemple : résolution du problème de conversion C-F....9
I.D- Remarques sur les exemples dans ce document 10
II- GESTION DES CONTRAINTES : UN EXEMPLE (DANS N)....11
II.A- Rappel du modèle utilisé (problème CSP)....13
II.B- Un exemple : la découpe....14
III- TECHNIQUES DE SATISFACTION DE CONTRAINTES....15
IV- SCHÉMA CLP(X) 16
IV.A- Le rôle des contraintes dans un CLP (et CSP)....17
V- CSP MULTIDISCIPLINAIRE 18
VI- QUELQUES EXEMPLES D'APPLICATIONS (REAL WORLD)....19
VII- COMPLEXITÉ DES PROBLÈMES ABORDÉS....24
VIII- NOTION DE CONTRAINTE-RÉPONSE....25
VIII.A- Un exemple de contrainte-réponse en N....27
IX- X DANS CSP(X), AUTRE QUE N, R, Z, Q 29
IX.A- Contraintes globales 29
IX.B- Contraintes sur les chaines....29
IX.C- Contraintes sur les arbres....29
IX.D- Domaines ensemblistes....30

X- CONTRAINTES RÉALISABLES PAR L'UTILISATEUR 31

X.A- Contraintes symboliques 31
X.B- Contraintes temporelles (réalisables par l'utilisateur)....................................32
X.C- Cas des contraintes Booléennes 33
X.C.1- Contraintes booléennes : un exemple de détection de pannes 34
X.C.2- un autre exemple Booléen : Dieu !....................................36
X.D- Calcul en arithmétique non linéaire....................................38

XI- UNE COMPARAISON AVEC LA PROGRAMMATION IMPÉRATIVE....................................40

XI.A- Comparaison avec Prolog (via un exemple)....................................41

XII- FORMALISATION DES CSP PAR DES EXEMPLES....................................48

XII.A- Formalisation de 8-reines....................................48
XII.A.1- Une autre formalisation de N-reines 50
XII.B- Construction de pont....................................52

XIII- RETOUR SUR LES SCHÉMAS CP (V, D, C) 54

XIII.A- Structure d'un CP (V, D, C) 54
XIII.B- Quelques types de contraintes 55
XIII.C- Exemple d'expressions de contraintes par l'utilisateur....................................56

XIV- LES CONTRAINTES DANS GPROLOG (ET GPROLOG-RH)....................................58

XV- CLP PAR DES EXEMPLES....................................61

XV.A- Exemple : têtes et pattes....................................61
XV.B- Rappel : conversion C/F revisitée (dans Q et R)....................................62
XV.C- Un exemple en Logistique....................................63
XV.D- Exemple des pierres (stone)....................................64
XV.E- Circuit électronique (Booléen) 67
XV.F- Car sequencing 69

XV.G- Détente : Casse-tête logiques (Lewis caroll)....72
XVI- (MÉTA) HEURISTIQUES UTILISÉES....76
XVI.A- Techniques de Parcours de graphes dans les CSP....76
XVI.A.1- Le principe First Fail....76
XVI.A.2- La technique Forward-Check....77
XVI.A.3- Simulation Forward-Check....79
XVI.B- Techniques Branch & Bound (séparation-évaluation)....80
XVI.B.1- Efficacité de l'heuristique B&B....84
XVII- NOTION DE HIÉRARCHISATION DES CONTRAINTES....85
XVII.A- Exemple indicatif....86
XVIII- ANNEXES....87
XVIII.A- Quelques notions sur les graphes....87
XVIII.B- Complément sur la CSP et CLP....91
XVIII.B.1- X dans CSP(X)....91
XIX- INTÉRÊTS DE LA CSP (ET DE LA CLP)....95
XIX.A- Paradigmes "Algorithmique" vs. "Contrainte"....97
XIX.B- Classes de problèmes (variés) résolus en CSP....98
XIX.C- Quelques Domaines d'application....101
XIX.C.1- Quelques Catégories d'applications....102
XIX.D- Éléments de programmation avec contraintes....105
XIX.D.1- Objectifs....105
XIX.E- Langages et environnements spécifiques de CP....106
XIX.E.1- Environnements de programmation spécifiques....106
XIX.E.2- Environnements génériques de programmation par contraintes....107
XIX.E.3- Environnements CLP....108
XIX.E.4- Environnements COP....108

XIX.F- X dans CSP(X)....109
XIX.G- Problème de reconnaissance de Scènes (Scene-labeling)....112
XIX.H- Raisonnement Temporel....121
XIX.I- Exercice : Énigme policière (relations entre événements)....125
XIX.J- D'autres exemples d'utilisation de CSP 126
XIX.J.1- Planning & Scheduling 126
XIX.J.2- Traitement des langues naturelles (TLN)....126
XIX.J.3- Bases de données (BD)....126
XIX.K- Isomorphisme de graphes....127
XIX.L- Solution au Problème (réel) de construction d'un Pont....128
XIX.L.1- Dessin du problème de pont....132
XIX.M- Exemple Bin Packing 134

XX- LE MODÈLE CONCURRENT POUR L'EXÉCUTION DE CSP (DOMAINE FINI)....135

XXI- LA COMPLEXITÉ DES PROBLÈMES CSP (DOMAINE FINI)....136

XXI.A- Le problème des N-reines....136

XXII- ÉLÉMENTS THÉORIQUES DU CSP....137

XXIII- DÉFINITIONS ET TERMINOLOGIES....137

XXIII.A- Notion de satisfaction 139
XXIII.B- Représentation des contraintes....142
XXIII.C- Transformation n-aire vers binaire....144
XXIII.D- Propagation des contraintes....146
XXIII.D.1- Consistance de Noeud (Node Consistancy)....147
XXIII.D.2- Consistance d'Arcs (Arc consistancy)....147
XXIII.D.3- Consistance de Chemin (Path consistancy) ou la K-consistance....148
XXIII.E- Caractéristiques de la CLP....150

I- Introduction 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 Gnu-Prolog (utilisé dans les TDs/TPs),
quelques autres permettent de compléter cet aperçu par des possibilités de résolution à l'aide des contraintes.

I.A- Le paradigme de Satisfaction de Contraintes (CSP)

Contraindre = maintenir dans des limites s'exercent sur les variables (paramètres)

<u>Genèse</u> : Affectation vs. Équation

- Test vs. Contrainte

- Ensemble de tests et d'affectations vs. un système d'équations / inéquations

En C / C++ (impératif)	*En Prolog*	*En CLP*
if (X > 0) Y = X+1;	***X > 0 , Y = X +1***	***X #> 0, Y #= X +1***
Si X>0, l'affectation a lieu...	(X > 0) /\ (Y = X+1) (X =< 0) /\ Y inchangé	On contraint X et Y X est contraint (X=Y-1)

➠ Statut et valeur <u>initiale</u> de X ?

➠ notion <u>d'attente</u> ?

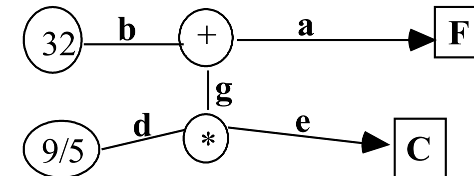
I.B- Une Idée intuitive (exemple de conversion F-C)

C = (F-32) * 5/9 ➠ F = 32 + 9/5 * C

Les tous premiers systèmes (*Steele*) dans une vision différentes 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.

I.C- Exemple : résolution du problème de conversion C-F

Conversion C°/F° : modélisation

- 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**

➜ On soumet le système suivant (PIII, 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)
    X in R
    Simplex

- Domaines : Réels (intervalles), Bool, Rationnels, Symboliques, Chaines, Ensembles, ...

I.D- Remarques sur les exemples dans ce document

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

La version basique de *Gprolog* permet la manipulation des contraintes dans $\mathbb{N}$, mais pas $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{Z}$.

Si vous souhaitez disposer d'un environnement avec plus de possibilités, GProlog-**RH**, **Bprolog** ou **Eclipse** (tous libres) peuvent bien convenir.

- 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}
```

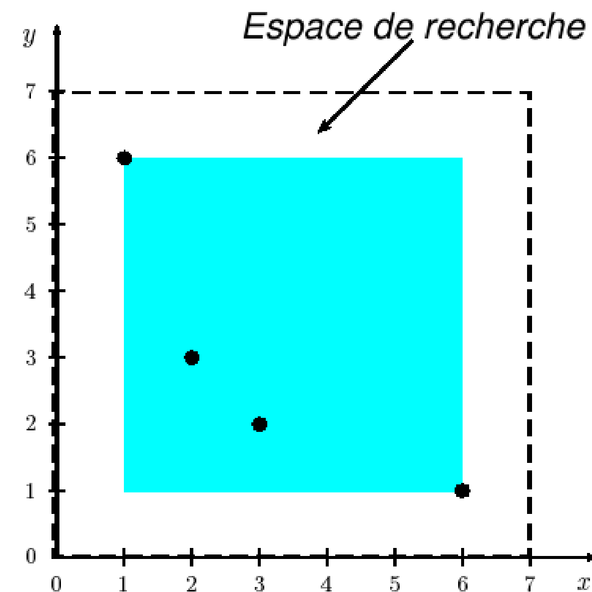
II- Gestion des contraintes : un exemple (dans N)

Soient les contraintes du problème suivant (dans 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

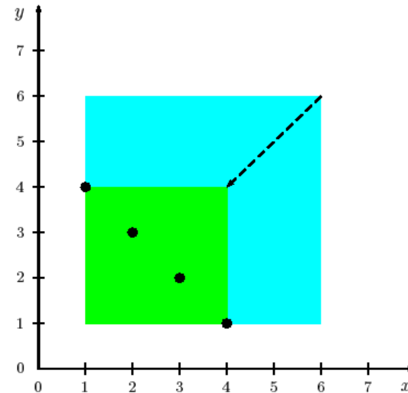
1) X in 0..7,
2) Y in 0..7,
3) X * Y = 6
4) X + Y = 5
5) X < Y.



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

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

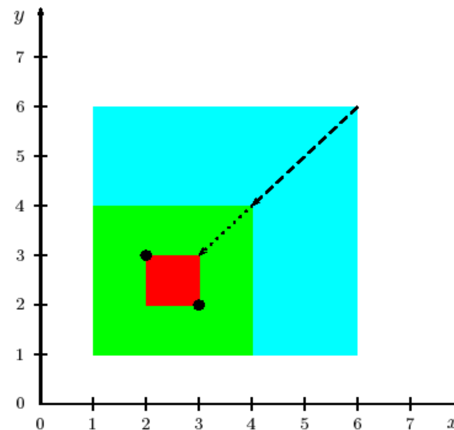
1) X in 0..7,
2) Y in 0..7,
3) X * Y = 6
4) X + Y = 5
5) X < Y.



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

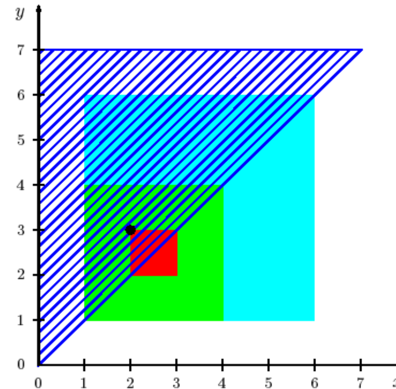
X in 2.. 3 , Y in 2.. 3

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 $\mathbb{Q}$: $\{X < 5 / 2,\ Y = 5 - X\}$

II.A- Rappel du modèle utilisé (problème CSP)

Étant donné $\mathbf{C(x_1 , . . . , x_n)}$ une contrainte sur des variables
$\mathbf{V=\{x_1 , . . . , x_n\}}$, avec $\mathbf{x_1 \in D_1 , . . . , x_n \in D_n}$,

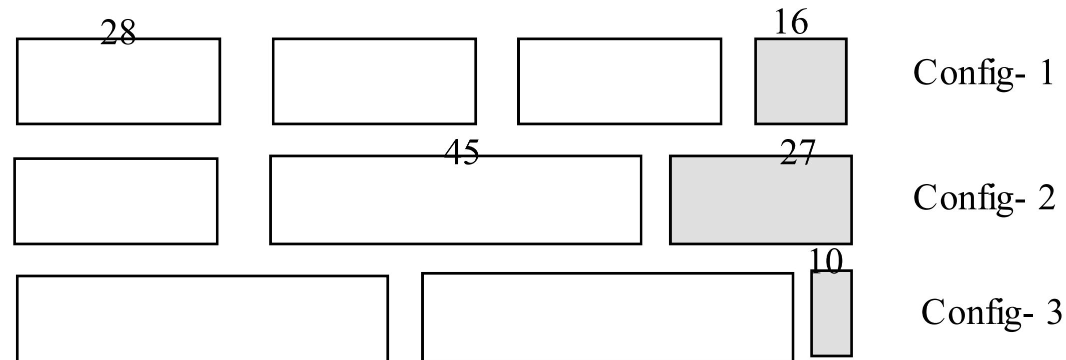
On appelle :

- **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 $\mathbf{(a_1 , . . . , a_n)}$ tels que
  $\mathbf{a_1 \in D_1 , . . . , a_n \in D_n}$ et $\mathbf{C(a_1 , . . . , a_n)}$ est vraie.

II.B- Un exemple : 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



Gnu Prolog :

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

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

III- Techniques de satisfaction de contraintes

- **Node/Arc/Path consistancy :**

Vérifications + un moteur de retour arrière (BT)

- ***X in R*** **:**

La vérification du domaine de X dans l'équation **X = Y +Z** est exprimée par

```
X = Y +Z  :
   X   in   min(Y)+min(Z) .. max(Y)+max(Z).     % X in R est une contrainte de base
```

➘ Cette vérification a lieu à chaque fois qu'une des 3 variables prend une valeur
ou si le domaine de l'une d'entre elles se modifie (suivi d'éventuelles réductions des autres domaines)

➘ La résolution de l'équation n'a lieu que lorsque 2 des 3 variables a reçu une valeur.
Sinon, la réponse = la forme la plus simplifié (avec domaines purgés) est présentée.

- ***Simplex*** **:**

Pour les systèmes d'équation-inéquation-Diséquation
Simplification Gausse-Jordan (seulement pour l'égalité)

Une combinaison de ces techniques est possible (Gprolog-RH, Bprolog et LP**)**

IV- Schéma CLP(X)

- CLP permet une *spécification déclarative* des problèmes; 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

IV.A- Le rôle des contraintes dans un CLP (et 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

- Une contrainte est déclarative : indépendante du processus de calcul

- Une contrainte n'est pas orientée, elle spécifie une relation :

  Dans $\boldsymbol{X + Y = Z}$ **,** 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 :

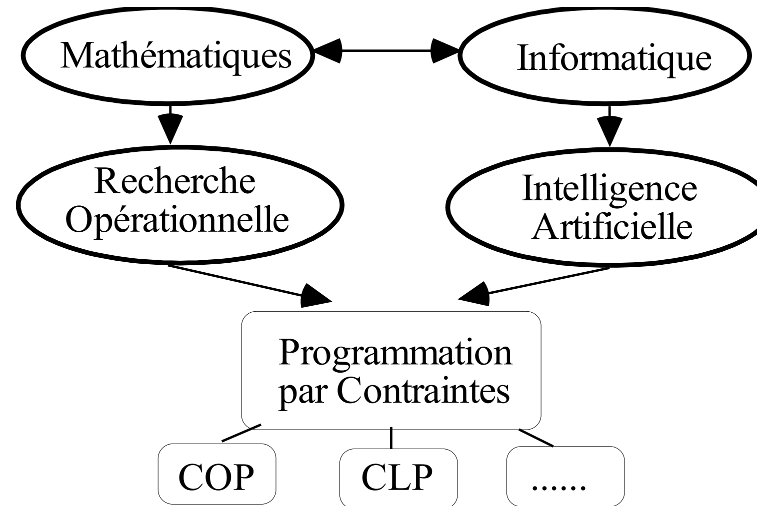
  Dans $X^2 = 2$

  - 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.

V- 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

VI- Quelques exemples d'applications (*real world*)

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

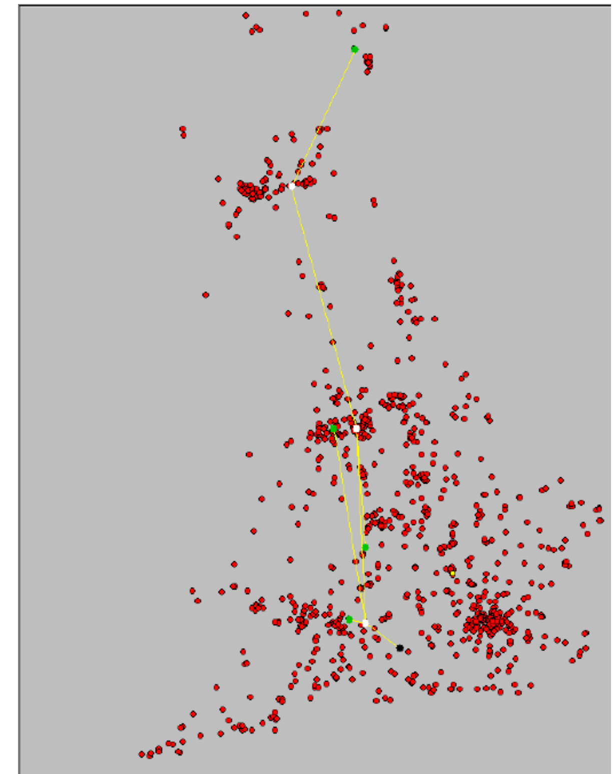
- emploi du temps
- construction d'horaires de personnel :
  - santé, commerce, usine, ...
- planification des équipages sur les avions, les trains
- tournée de véhicules
- ...

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



Retombés attendues :
Stratégie : quel client servir (contrat ?)
Profit : combien et comment charger (tel trajet n'apporte pas / beaucoup / …)

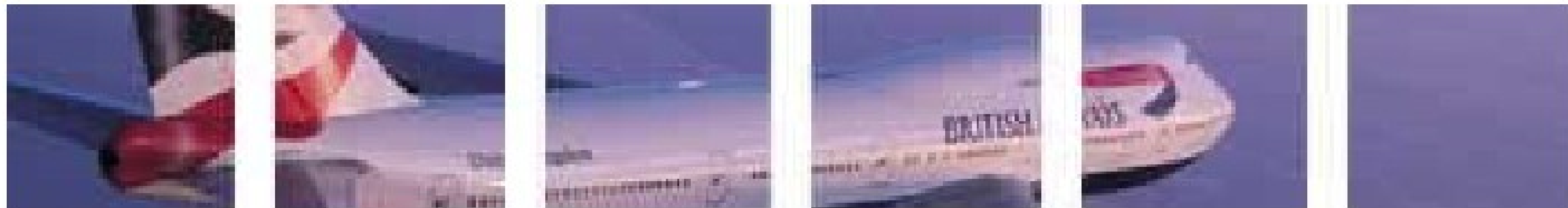
Flight Schedule Retimer :

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

VII- Complexité des problèmes abordés

Résolution de problèmes combinatoires (NP-Complet)

<u>Exemples de complexité :</u>

1 • Crypt-arithmétique (S,E,N,D,M,O,R,Y $\in$ {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

2 • 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

	1	2	3	4	5	6	7
A				■			
B							
C			■		■		
D	■						■
E			■		■		
F							
G				■			

Un exemple de grille 7 x 7

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

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 !

VIII- 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.

Exemple (en GProlog-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).
```

../..

Questions :

1- Quelle mensualité pour un capital = 100 sur 12 mois, taux=10% (on ne doit rien à la fin) :

versement(100, 12, 0.1, 0, M). => M = 14.67633151

2- Combien emprunter et quelle est la mensualité sur 12 mois à un taux annuel de 10% pour que l'on doive capital /2 à la fin.

versement(C, 12, 0.1, C * 0.5 , M). => {M = 0.12338165755 * C }

2 bis - Pour transformer la contrainte réponse en une réponse :

- Et si on rembourse 15 euros par mois ?
  versement(C, 12, 0.1, C*0.5, 15). => C = 121.57398675

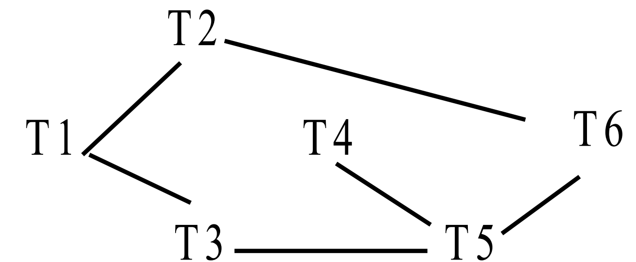
- Et si on emprunte 150 euros ?
  versement(150, 12, 0.1, C*0.5, M). => M = 18.507248633

3- Combien emprunter pour une mensualité = 15 (sur 12 mois, taux=10%, reste = 0) :
versement(P, 12, 0.1, 0, 15). => P = 102.20537734

VIII.A- Un exemple de contrainte-réponse en N

Ex. d'ordonnancement (PERT) :

Graphe de précédence de tâches et ordonnancement



- Une tâche placée à droite s'effectue après celles à sa gauche.
- Durée de chaque tâche = 1 heure
- Début de chaque tâche ∈{1,2,3,4,5}
- Les contraintes :

```
avant(T1, T2).        % T1 #< T2
avant(T1, T3).
avant(T2, T6).
avant(T3, T5).
avant(T4, T5).
avant(T5, T6).
diff(T2, T3).             % T2 et T3 disjonctives #\=
```

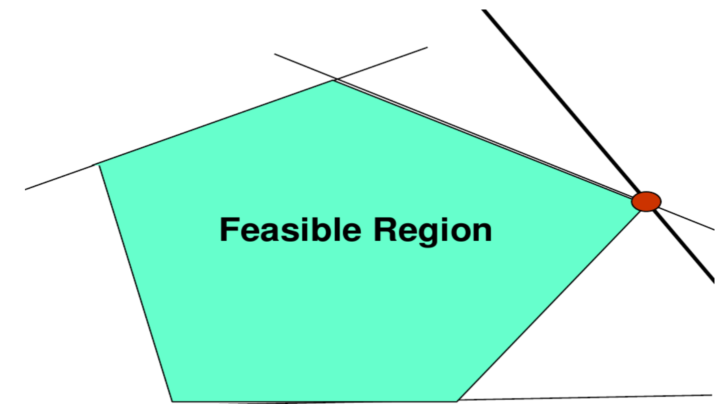
Consistance et de **propagation des contraintes :**

- Avant tout calcul, les propagations successives réduisent les domaines et donnent la solution générale :

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

- On procède ensuite par énumération (les domaines sont finis)
  Par exemple : T1 = 2 ➔ T2=4, T3=3, T4∈{1,2,3}, T5=4, T6=5.

- Si l'on est contraint à terminer toutes les tâches en 4 heures :
  - ➔ Max(Ti) <=4
  - ➔ T6∈{4,5} ➔ T6=4
  - ➔ T5<T6, T5∈{3,4} ➔ T5=3
  - ➔ T4 < T5 , T4 ∈{1,2,3}, T5=3 ➔ T4 ∈{1,2}
  - ➔ T3 < T5 , T3∈{2,3} , T5=3 ➔ T3=2
  - ➔ T1 < T3 , T1∈{1,2} , T3=2 ➔ T1=1
  - ➔ T2∈{2,3,4}, T2 < T6, T6=4 ➔ T2 ∈ {2,3}



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

IX.A- 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 :

IX.B- Contraintes sur les chaines

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

IX.C- Contraintes sur les arbres

IX.D- Domaines ensemblistes

Manipulation des ensembles (opérateur /\ )

```
?- 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

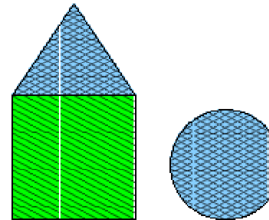
X- Contraintes réalisables par l'utilisateur

X.A- Contraintes symboliques

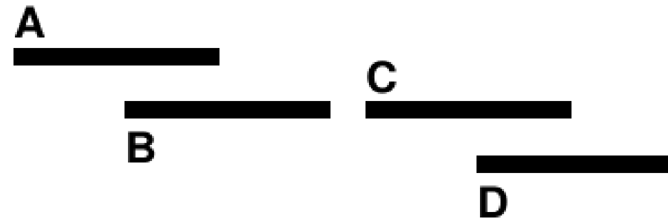
- Les contraintes sont exprimées sur des arbres (comme pour l'unification Prolog).
  Deux arbres (termes composés) seront identiques si leurs composants le sont.

- On peut manipuler des contraintes symboliques :

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



X.B- Contraintes temporelles (réalisables par l'utilisateur)



De :

rel(AoverlapB, BbeforeC, R_AC)
& *rel(BbeforeC,CoverlapD, R_BD)*
& *rel(R_AC, CoverlapD, R_AD)*

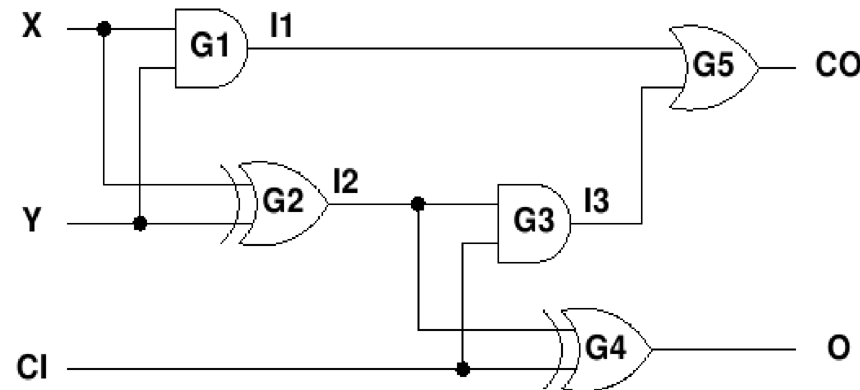
On obtient :

R_AC= AbeforeC
& *R_BD=BbeforeD*
& *R_AD=AbeforeD*

Voir plus loin pour les détails des contraintes temporelles.

X.C- Cas des contraintes Booléennes

Exemple additionner



Modélisation :

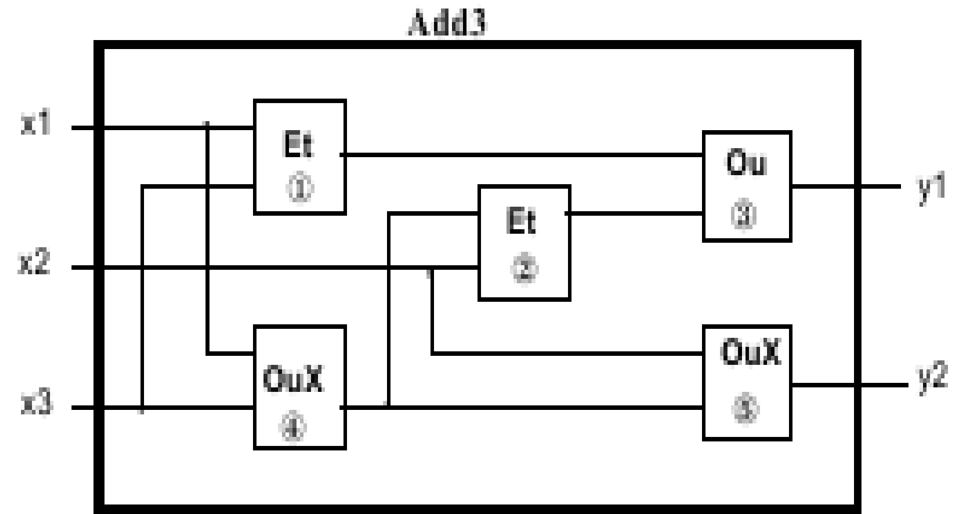
Variables : entrées/sorties des portes
Domaines : [0,1]
Contraintes Booléennes (pseudo code) :

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

Ou encore :

```
and(X, Y, I1), xor(X, Y, I2), and(I1, CI, I3),
xor(I2, CI, O), or(I1, I3, CO).
```

X.C.1- Contraintes booléennes : un 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 *AuPlusUnVrai*.

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.

Une solution

Solution spécifique à ce circuit.

```
circuit([X1,X2,X3], [Y1,Y2], [P1,P2,P3,P4,P5]) :-
    #\P1 #==> ( U1 #<=> X1 #/\ X3 ),
    #\P2 #==> ( U2 #<=> X2 #/\ U3 ),
    #\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

X.C.2- 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

```
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.

X.D- Calcul en arithmétique non linéaire

Exemple : arithmétique non-linéaire

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

- 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;

Exemple (GProlog)

```
?-  fd_set_vector_max(5000). fd_domain([T,V,D],1,500),
    D = V * T, D=(V-20) * (T+3) , D=(V+20) * (T-2).
```

==> Le système d'équations simplifié :

{3V - 20 T - 60 = 0,
-2V + 20 T - 40 = 0}

Si V=100 :

```
?-  D = V * T, D=(V-20) * (T+3) , D=(V+20) * (T-2), V=100.
```

V= 100 Km/h
T = 12 minutes
D = 100 * 12/60 = 20 Km

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.
```

Question :

```
?- zmul(1, 2, R2, I2, R3, I3).
     ==>     I2 = 0.2 * I3 - 0.4 * R3
             R2 = 0.4 * I3 + 0.2 * R3
```

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

XI- Une comparaison avec la programmation impérative

programmation logique	**programmation impérative**
règle	procédure
ensemble de règles	programme
question (but)	appel de procédure
preuve	exécution
substitution, unification	passage de paramètres

XI.A- Comparaison avec Prolog (via un exemple)

Quelques comparaisons entre Prolog (std) et Prolog avec Contraintes.
Ces exemples montrent les différences entre les deux domaines dans diverses situations.

1) Différences en Arithmétique simple

Approche Prolog :

```
    p(X,Y,Z):- Z is X+Y.
      :- p(3,4,Z).        => Z=7
      :- p(X,4,7).        => INSTANTIATION ERROR
```

Approche CLP :

```
    p(X,Y,Z):- Z #= X+Y.
      :- p(3,4,Z).        =>  Z=7
      :- p(X,4,7).        =>  X = 3.
```

1 bis) Exemple factoriel

Approche Prolog :

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

:- fact(4, X).   ==> X= 24.
:- fact(X, 24) , !.==> INSTANTIATION ERROR

Approche CLP :

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

:- fact(4, X).   ==> X= 24.
:- fact(24, X) , !.==> X=4

N.B. : la présence de '!' est nécessaire, sinon, la recherche d'une seconde solution est sans fin (comportement normal).

2) Différence dans la Méthode

- L'arithmétique n'est pas **relationnelle** en Prolog Std.

→ **générer et tester** est la seule approche possible (avec des variantes plus ou moins efficaces) :

```
solution(X,Y,Z):-
        p(X), p(Y), p(Z),
        test(X,Y,Z).

p(11).        p(3).        p(7).
p(16).        p(15).       p(14).

test(X,Y,Z):-
        Y is X+1, Z is Y+1.
```

:- solution(X,Y,Z). => 458 pas (étapes) pour la première solution : env > 6 x 6 x 6 x 2

Il n'est pas possible de placer l'appel à test avant p/1 :

```
solution(X,Y,Z):-
        test(X,Y,Z), p(X), p(Y), p(Z).
```

Provoquera des erreurs d'instanciation (à cause de **is**)

- L'arithmétique relationnelle en CLP.

→ la méthode conseillée : ***contraindre et générer*** :

```
solution(X,Y,Z):-
        test(X,Y,Z),            % générer
        p(X), p(Y), p(Z).

p(11).          p(3).       p(7).
p(16).          p(15).      p(14).

test(X,Y,Z):-
        Y #= X+1, Z #= Y+1.
```

:- solution(X,Y,Z).
=> seulement 11 pas pour la première solution

De plus, si des domaines sont définis pour X,Y et Z, on améliorera le nombre d'étapes.

3) Modélisation (avec trace) :

```
test1(X,Y) :- p(X), q(Y), Z is X+Y, Z < 10.        %générer-Tester

test3(X,Y) :-                                       % Contraindre-générer
   Z #=# X+Y, Z #<# 10, p(X), q(Y).

/* Pas possible d'écrire la même chose pour test1 */
p(5).     p(7).     p(10).     p(8).
q(5).     q(7).     q(3).      q(8).
```

La différence de traitement :

```
| ?- test1(A,B).
   1    1  Call: test1(_16,_17) ?
   2    2  Call: p(_16) ?
   2    2  Exit: p(5) ?
   3    2  Call: q(_17) ?
   3    2  Exit: q(5) ?
   4    2  Call: _146 is 5+5 ?
   4    2  Exit: 10 is 5+5 ?
   5    2  Call: 10<10 ?
   5    2  Fail: 10<10 ?
   3    2  Redo: q(5) ?
   3    2  Exit: q(7) ?
   4    2  Call: _146 is 5+7 ?
```

```
4   2  Exit: 12 is 5+7 ?
5   2  Call: 12<10 ?
5   2  Fail: 12<10 ?
3   2  Redo: q(7) ?
3   2  Exit: q(3) ?
4   2  Call: _146 is 5+3 ?
4   2  Exit: 8 is 5+3 ?
5   2  Call: 8<10 ?
5   2  Exit: 8<10 ?
1   1  Exit: test1(5,3) ?
```

A = 5
B = 3 ? ;   => trace longue, pas d'autres réponses

Et dans la cas des contraintes :

test3(A,B). **=> Remarquez les domaines dans les requêtes.**

```
1   1  Call: test3(_16,_17) ?
2   2  Call: _98#=#_16+_17 ?
2   2  Exit: _#41(0..268435455)#=#_#3(0..268435455)+_#22(0..268435455) ?
3   2  Call: _#41(0..268435455)#<#10 ?
3   2  Exit: _#41(0..9)#<#10 ?
4   2  Call: p(_#3(0..9)) ?
4   2  Exit: p(5) ?
5   2  Call: q(_#22(0..4)) ?
5   2  Exit: q(3) ?
1   1  Exit: test3(5,3) ?
```

A = 5
B = 3 ? ; => on peut demander d'autres solutions. Réponse rapide.

```
1   1  Redo: test3(5,3) ?
5   2  Redo: q(3) ?
5   2  Fail: q(_#22(0..4)) ?
4   2  Redo: p(5) ?
4   2  Exit: p(7) ?
5   2  Call: q(_#22(0..2)) ?
5   2  Fail: q(_#22(0..2)) ?
4   2  Redo: p(7) ?
4   2  Exit: p(8) ?
```

Suite ↗

```
5   2  Call: q(_#22(0..1)) ?
5   2  Fail: q(_#22(0..1)) ?
1   1  Fail: test3(_16,_17) ?
no
```

Comparer les deux traces !

XII- Formalisation des CSP par des exemples

- Il est important de savoir formaliser un CSP
- Elle permet d'exprimer les contraintes réduisant les domaines
- Il peut y avoir plusieurs formalisations possibles pour un CSP

	A	B	C	D	E	F	G	H
1								
8								

XII.A- Formalisation de 8-reines

Identification des variables et leur domaines :

- Chaque ligne représente une variable : $Z=\{Q_1,\ldots,Q_8\}$
- Le domaine de chaque variable est une colonne :

$D_{Q_1} = D_{Q_2} = \ldots. = D_{Q_8} = \{1..8\}$

- 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* $I-J\neq b-a$

<u>Exemple</u> : $Q_1=1,\ Q_2 = 3$

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
```

- Pour N=8, il y a 8^8 combinaisons.

- **Formaliser les contraintes :**
  exprimer “sur la même diagonale”
  (puis utiliser sa négation)

XII.A.1- Une autre formalisation de N-reines

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

  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 !
  En fait, dans le cas précédent, la contrainte C1 était supprimée.
- Cet exemple montre l'importance de la phase d'analyse d'un CSP.

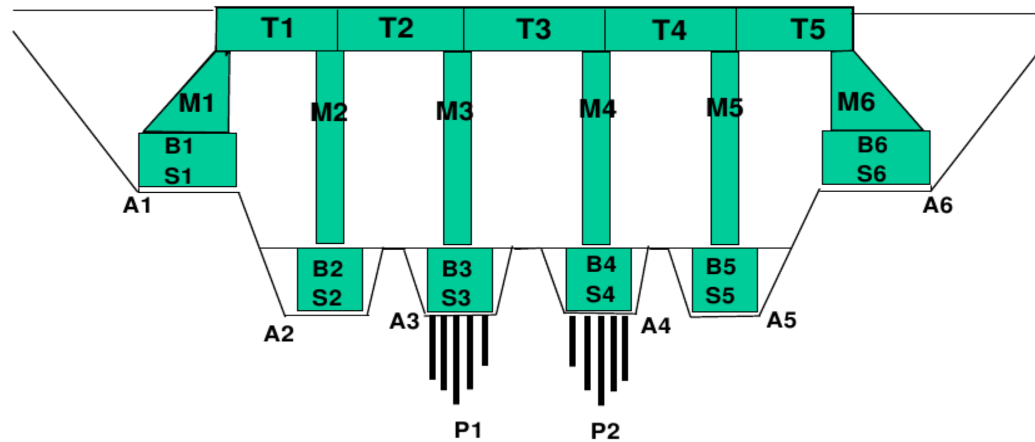
- La formalisation booléenne est également possible (2^{64}).

- 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 pas présents que dans peu de CSP.

XII.B- 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 :

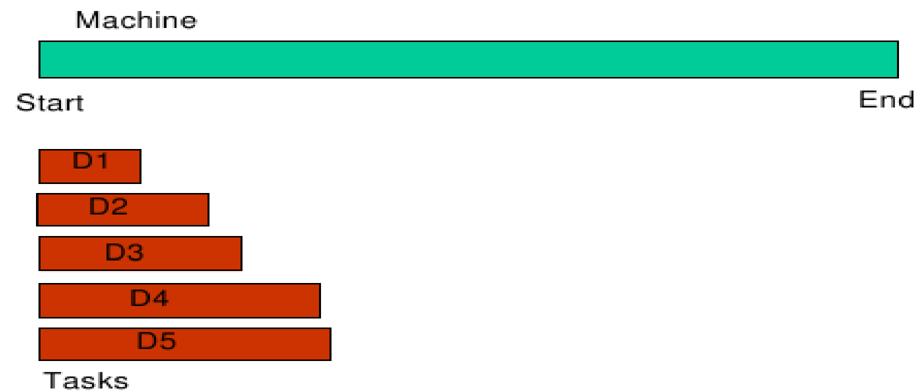
  before(Start1,Duration1,Start2)

- 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).

XIII- Retour sur les schémas CP (V, D, C)

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.

XIII.A- Structure 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 =>

XIII.B- Quelques types de contraintes

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

- Contraintes sur les domaines : ***fd_domain(B, 2, 25)***
  fd_domain([X,Y], [1,3,5,7])

- Contraintes arithmétiques: ***A #= B, A #> B, A #\= B***
  (avec plus de pré traitement) ***A #=# B, A #>># B, A #\=# B***

- Contraintes symboliques ***fd_atmost(N, Liste, Val)*** ***fd_element(Index, Liste, Val)***

- Contraintes booléennes: ***A*** #\/ ***B, A #==> B, A #⇔B***

- Contraintes d'évaluation ***#=(A, B, C: Bool)*** ***C #<=> (A=B)***
  #>=(A, B, C:Bool) #\/ ***(A, B, C:Bool)***

- Contraintes globales (niveau plus élevé de consistance) ***fd_all_different([X1,...,Xn])*** $X_i \neq X_j$, $i \neq j$
  fd_cardinality(Liste, Nbr)

Exemple :

fd_all_different([X1, X2, X3]) , fd_domain([X1, X2, X3], 1, 2). ==> non.

Les incohérences/simplifications peuvent être détectées par avance.

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

XIII.C- Exemple d'expressions de contraintes 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 <u>qu'une fois</u>;
- Il peut visiter 2 sites en deux jours consécutifs ou espacés d'au plus 1 jour.

<u>Expression des contraintes : un exemple</u>

- 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).
  ==> ***fd_domain(Liste_des_sites, 1, 7)***

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

- Visiter chaque site une seule fois : ***fd_all_different(Liste_des_sites)***

- Pour toute paire de sites S_1 et S_2 :
  S1 #= S2 +X, fd_domain(I,1,4), fd_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 (Eclipse 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.

XIV- Les contraintes dans Gprolog (et GProlog-RH)

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

Finite Domain variables

FD variable parameters

fd_max_integer/1
fd_vector_max/1
fd_set_vector_max/1

Initial value constraints

fd_domain/3, fd_domain_bool/1
fd_domain/2

Type testing

fd_var/1, non_fd_var/1, generic_var/1, non_generic_var/1

FD variable information

fd_min/2, fd_max/2, fd_size/2, fd_dom/2
fd_has_extra_cstr/1, fd_has_vector/1, fd_use_vector/1

Arithmetic constraints
FD arithmetic expressions

Partial AC: mise à jour des bornes des domaines

(#=)/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

Full AC: mise à jour de la totalité de chaque domaine

(#=#)/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
fd_prime/1, fd_not_prime/1

Boolean and reified constraints

(#\)/1 - constraint NOT,
(#<=>)/2 - constraint equivalent,
(#\<=>)/2 - constraint different,
(##)/2 - constraint XOR,
(#==>)/2 - constraint imply,
(#\==>)/2 - constraint not imply,
(#/\)/2 - constraint AND,
(#\/\)/2 - constraint NAND,
(#\/)/2 - constraint OR,

(#\/)/2 - constraint NOR

fd_cardinality/2, fd_cardinality/3, fd_at_least_one/1, fd_at_most_one/1, fd_only_one/1

Symbolic constraints

fd_all_different/1
fd_element/3
fd_element_var/3
fd_atmost/3,
fd_atleast/3,
fd_exactly/3
fd_relation/2,
fd_relationc/2

Labeling constraints

fd_labeling/2, fd_labeling/1, fd_labelingff/1

Optimization constraints

fd_minimize/2, fd_maximize/2

A propos de fd_set_vector_max(5000) :

En Gprolog, le nombre de valeurs différentes pour une variable est limité par défaut à 127 (obtenu pas fd_vector_max(X)). Ce qui est assez limité mais ne consomme pas beaucoup de ressources.

La requête **fd_set_vector_max(N)** demande à étendre cette possibilité à N. En contre partie, les simplifications des domaines prennent plus de temps.

XV- CLP par des exemples

XV.A- Exemple : têtes et pattes

Calculer le nombre de pattes des chevaux (C) et des pigeons (P),
P et C sont des entiers, + et * ont leur sens arithmétique.

Spécification d'une relation entre P et C :

Syntaxe Gprolog :

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

?- nombre_de_tetes_et_de_pattes(6,2,U,V).
▶ U=8, V=20
?- nombre_de_tetes_et_de_pattes(X,Y,8,20).
▶ X = 6, Y = 2

XV.B- Rappel : conversion C/F revisitée (dans Q et R)

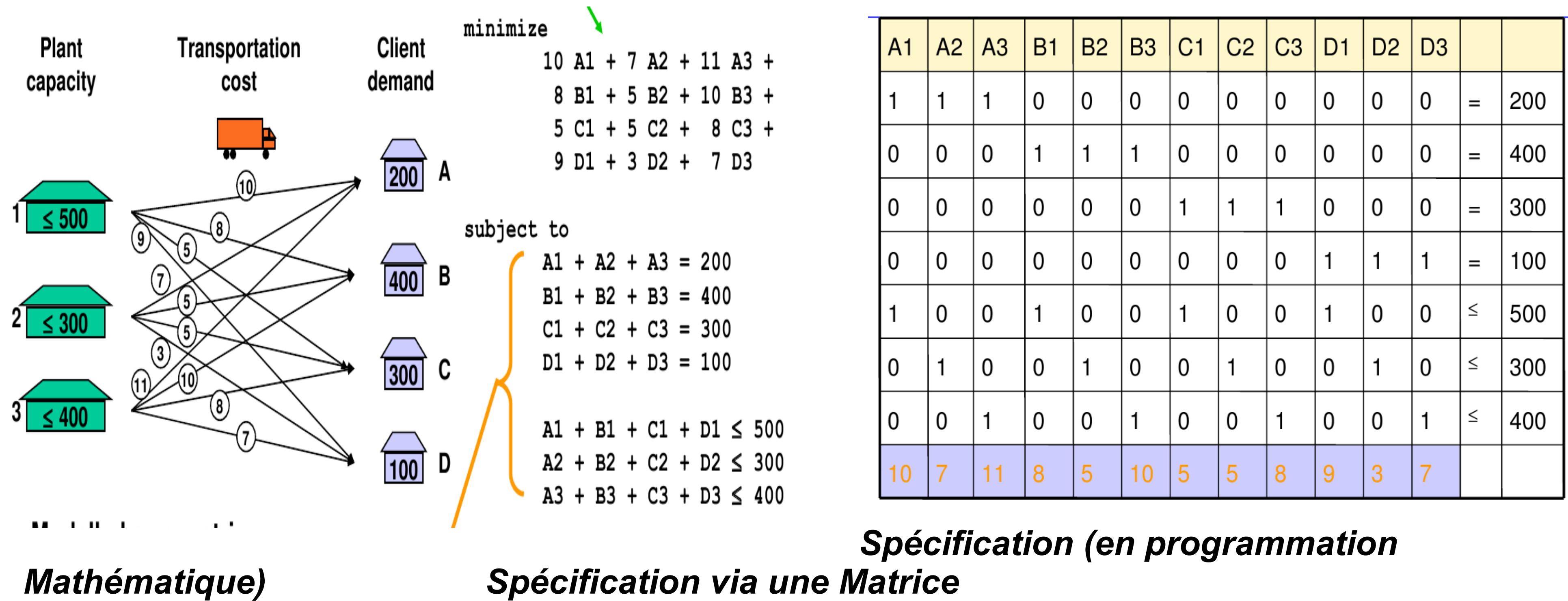
F = (9/5) C + 32, F=X*C+10, C+F=200 .

On obtient ➜ {F = 140, C = 60, X = 13/6}

Dans R : poser la question sous Eclipse par ($= : dans R) :

*F $= (9/5)*C+32, F $= X*C+10, C+F $= 200.*

XV.C- Un exemple en Logistique



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		

Spécification (en programmation Mathématique)

Spécification via une Matrice

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

XV.D- Exemple des pierres (stone)

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 ?

Découpage et spécification des contraintes :

Le problème est assez simple et n'a pas besoin d'un découpage sophistiqué.
On sait qu'il y aura 4 variables (**entiers**) pour les 4 morceaux P1, P2, P3 et P4 dont la somme doit être = 40.
Il semble également naturel que les 4 poids pourraient être différents.
L'expression des contraintes contiendra ici le découpage (les étapes).
On utilisera des **entiers** (les poids) dans le domaine 1..40

Le prédicat principal (contenant l'énumération finale) :

```
stone(L) :-
    L = [P1, P2, P3, P4],
    L :: 1.. 40,                      % notation générique du domaine
    P1 + P2 + P3 + P4 = 40,
    P1 =< P2, P2 =< P3, P3 =< P4,
    tous_les_poids_mesurables(L, 1, 40),
    énumérer(L).
```

NB : l'ordre établi entre *Pi* permet d'éviter des solutions symétriques.

Spécifier les relations :

à ne pas oublier que les relations entre les variables expriment également des contraintes.

```
tous_les_poids_mesurables(_, A, A) .          % on aura vérifié de 1 à 40
tous_les_poids_mesurables(L, A, B) :-         % vérifier pour A, puis de A+1 à B
    A < B ,
    peser(L, A),
    A1 = A+1,
    tous_les_poids_mesurables(L, A1, B).
```

Chaque poids peut être d'un côté ou de l'autre de la balance (ou inutilisée) :

```
peser(A, [P1, P2, P3, P4]) :-          % Vérifier que le poids A este mesurable par les Pi
    A = B1*P1 + B2*P2 + B3*P3 + B4*P4,
    [B1, B2, B3, B4]  ::  -1 .. 1.     % -1, 0 ou 1
```

Solution : **L = [1, 3, 9, 27].**

Remarque sur l'adaptation à l'outil Gprolog :

Selon le type des contraintes, on choisira le système CSP/CLP.
Un solveur linéaire pourrait-il faire l'affaire ? Comment faire pour *peser* ?
Avec GnuProlog en main qui traite des entiers positifs ou nul (et les réels).
Ce que Gprolog nous impose nous gène dans *peser* (pas d'entier négatif).

<u>Adaptation de peser</u> : changement de domaine : passer de *{-1, 0, 1}* à *{0, 1, 2}*

En Gprolog :

```
peser(A, [P1, P2, P3, P4]) :-
	fd_domain([B1,B2,B3,B4],0,2),
	A #=# (B1-1)*P1 + (B2-1)*P2 + (B3-1)*P3 + (B4-1)*P4.
```

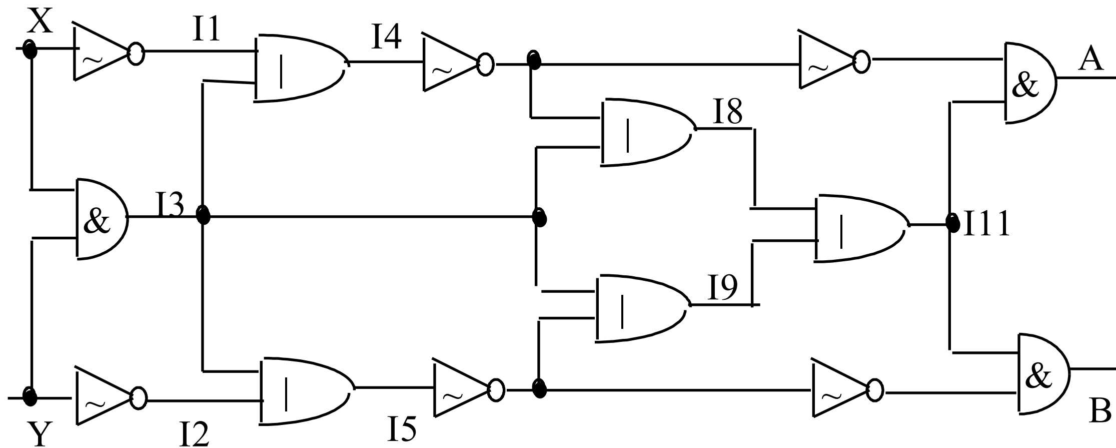
Solution : L = [1, 3, 9, 27].

Remarque :

Les autres prédicats s'adaptent sans difficulté à GnuProlog.
D'autres conventions sont possibles pour *perser*.
La vérification des propriétés est ici simple. Il n'y a pas d'optimisation particulière.

XV.E- Circuit électronique (Booléen)

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



N.B. : contraintes booléennes (Gprolog)

#/\ : et booléen
#\/ : ou booléen
#<=> : implication,
#\ : not
1 : true
0 : false

```
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}
```

Applications :

- Simplification, Optimisation, ...
- Simulation
- Détection de pannes dans les circuits.

XV.F- 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 <u>capacité</u> 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.

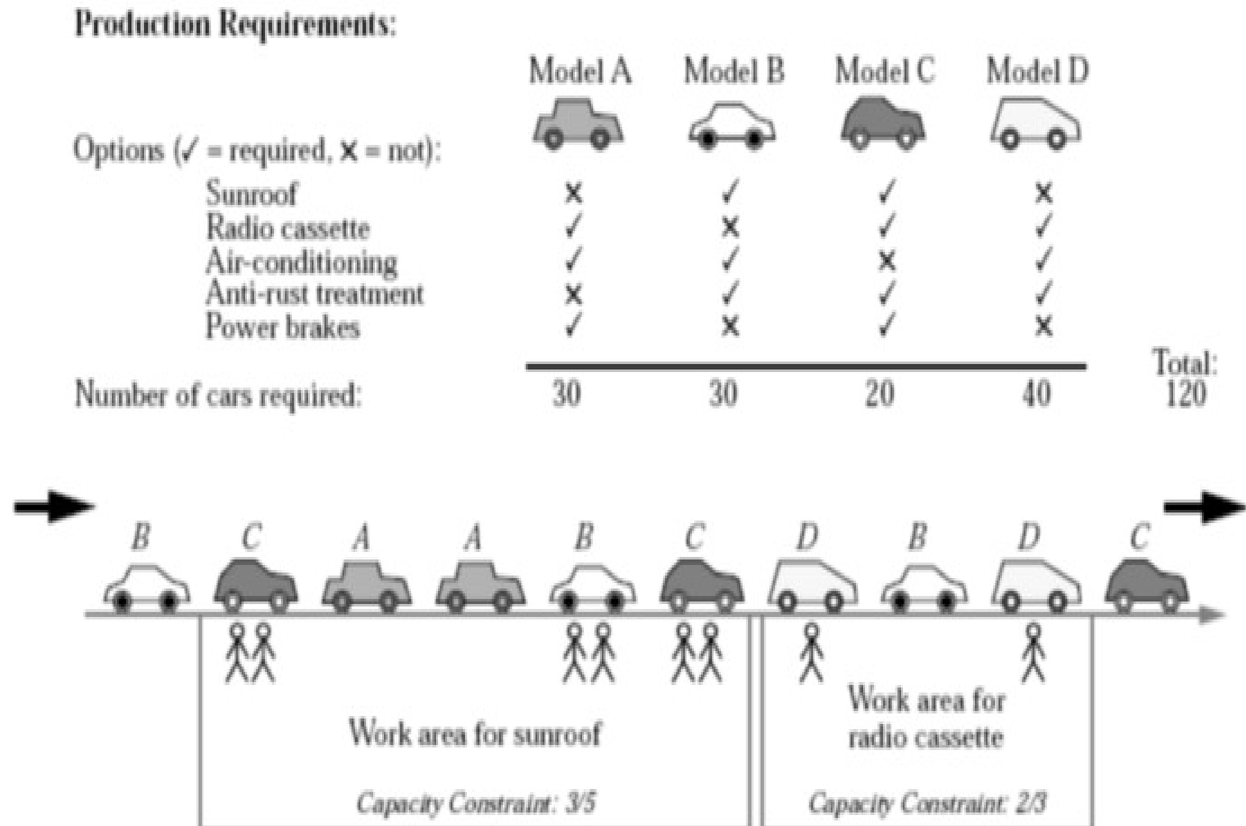
- Pendant ce temps d'installation, 5 autres voitures seront passées devant ce poste.
- Si 2 d'entre elles nécessitent le toit ouvrant, les deux autres équipes pourront les équiper.
- Toutefois, si une 4ème voiture nécessite cette même option, le poste ne sera pas assez rapide pour réagir à temps.

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.

L solution est présenté sur le schéma :

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

XV.G- 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 ? "

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">,                                   %affirmation 1
 <d,"employer son influence a des fins méritoires">,
 <e,"être affiche dans les vitrines">,
 <f,"être apte a être député">,                                %affirmation 1
 <g,"être un bienfaiteur du peuple">,                          %affirmation 1
 <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,                                           %affirmation 1
        (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

XVI- (Méta) Heuristiques utilisées

Voir en Annexes quelques notions sur les Graphes;

XVI.A- Techniques de Parcours de graphes 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 :

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

XVI.A.1- 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.

XVI.A.2- 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 :

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.
```

XVI.A.3- 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.
```

XVI.B- Techniques Branch & Bound (séparation-évaluation)

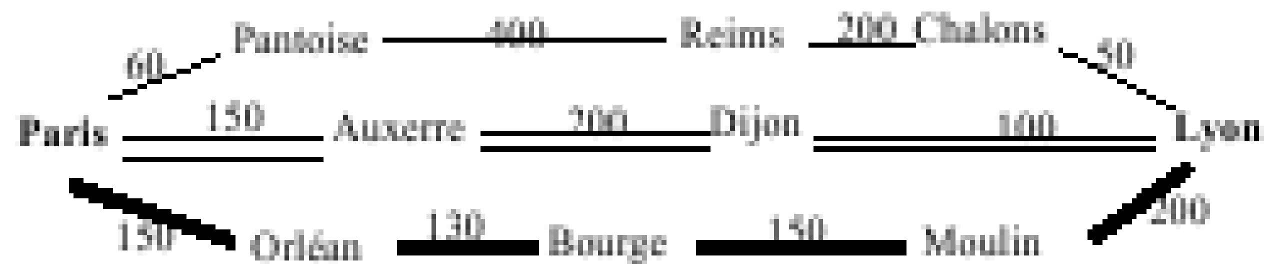
La technique générale **Branch-and-Bound** consiste à diviser le problème en plusieurs sous-problèmes (phase de séparation permettant de découper un coût global en sous coûts)
puis, dans la phase d'évaluation,
à trouver une évaluation de la borne du problème (par exemple le coût de la solution minimale/maximale 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 Gprolog, **fd_minimize** et **fd_maximize** permettent d'avoir accès à cette heuristique.

Exemple :

Les prédicats **p/1, q/1** et **r/1** produisent une combinaison de 27=3x3x3 solutions dont on veut le minimum.

```
p(3).        p(2).       p(1).
q(6).        q(5).       q(7).
r(12).       r(11).      r(10).

std_simple(T) :- p(A), q(B), r(C), T  is A+B+C .        % donne 27 solutions
fd_simple(T) :- T #= A+B+C, p(A), q(B), r(C).           % Idem
fd_BB(T) :- fd_minimize(fd_simple(T),T).                % B & B de GProlog

% B & B manuelle
manual_bb(T) :- fd_simple(Ref), suite(T,Ref).
suite(T, Ref) :- Sol #< Ref, fd_simple(Sol), !, suite(T,Sol).
suite(T, T).
```

Questions :

std_simple(T). => T=21, T=20, T=19, T=20, ...
fd_simple(T). => idem
fd_BB(T). => T=16
manual_bb(T). => T=16(il faudra couper).

NB : on peut compliquer en demandant que T soit borné (e.g. **std_simple(T), T < 20**) et comparer les efficacités des schémas.

Remarque sur les variables globales :

On peut implanter ces heuristiques en utilisant les variables globales en Gprolog (si l'on ne veut pas utiliser *fd_minimize*).

L'intérêt de cette implantation manuelle est d'avoir l'opportunité de placer librement la comparaison au coût de référence.

Cas de contraintes disjonctives :

L'utilisation des variables globales permet également de contrôler l'application des contraintes disjonctives qui sont très coûteuses en programmation sous contraintes.

En effet, la présence des contraintes disjonctives conduirait au développement d'un arbre de recherche pouvant déboucher sur un échec.
Il faudra alors remettre en cause un ensemble important de déductions et de calculs.

Une règle d'or est de Repousser l'utilisation des contraintes disjonctives au plus tard possible.

XVI.B.1- 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).          % Idem

% B & B manuelle
manual_bb(T) :- fd_simple(Ref), suite(T,Ref).
suite(T, Ref) :- Sol #< Ref, fd_simple(Sol), !, 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) :- g_assign(cout, Val).
lire_cout(Val) :- g_read(cout, Val).
maj_cout(Val) :- g_read(cout, Old_Val), (Val < Old_Val -> save_cout(Val) ; true).
```

?- bb_efficace(T). ==> T = 16

XVII- 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(----)
```

XVII.A- Exemple indicatif

Exemple indicatif (Gprolog-Rh) :

```
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.

XVIII- Annexes

XVIII.A- Quelques notions sur les graphes

- Graphe(G) = Un tuple(V,U)
  V : un ensemble de nœuds
  U $\subseteq$ (V x V) un ensemble d'*arcs*

- Un graphe *non orienté* = Un tuple(V,E)
  V : un ensemble de nœuds
  E $\subseteq$ (V x V) un ensemble d'*arêtes*

- Les nœuds dans un <u>arc</u> sont ordonnées
  Les nœuds dans une <u>arête</u> ne sont pas ordonnées

- Pour un arc (x,y), les arêtes sont (x,y) et (y,x).

- Deux nœuds d'une arête sont dits *adjacents* :
  Pour graphe(G) = (V,E), adjacent(x,y) ssi (x,y) $\in$ E

- Un CSP <u>binaire</u> peut être représenté (visualisé) par un graphe non orienté de contraintes dont les nœuds sont des variables et chaque <u>arête</u> représente une contrainte binaire.

Hyper-graphe et les CSP

- Arête : un couple de nœuds connectés
  Hyper-arête : un ensemble de nœuds connectés
  Hyper-graphe = un ensemble de nœuds et un ensemble d'hyper arêtes (c'est une généralisation de graphe).

- L'hyper-graphe H(P) d'un CSP(P)= (Z, D, C) est un hyper-graphe où les nœuds sont des variables dans Z et les hyper-arêtes représentent les contraintes dans C.

- Si le CSP est binaire alors H(P) est un graphe noté G(P).

Chemin (branche, path)

- Un ensemble de nœuds où toute paire de nœuds adjacents est connectée par une arête (ou un arc si le graphe est orienté) :
  $\text{chemin}((X_1,\ldots, X_k) , (V,E)) \equiv (X_1,X_2) \in E \wedge \ldots \wedge (X_{k-1},X_k) \in E$

- La *longueur* d'un chemin contenant N nœuds = N-1.
  Les nœuds du chemin ne sont pas forcément distincts.

- Un nœud Y est *accessible* depuis un nœud X s'il existe un chemin de X vers Y :
  $(X,Y) \in E \;\&\; (\exists Z_1,\ldots Z_k : \text{chemin}((\mathbf{X},Z_1,\ldots Z_k,\mathbf{Y}) , (V,E)))$

Graphe connecté (connexe)

- Un graphe où il existe un chemin entre toute paire de nœuds :
  pour tout $X,Y \in V$, accessible(X,Y,(V,E))

- Un graphe de contraintes peut ne pas être connexe (des variables non contraintes peuvent exister)

Boucle (loop) dans un graphe

- Il existe un arc ou une arête (x,x).
  Une boucle ne concerne qu'un nœud.

Réseau

- Un graphe connexe et <u>sans</u> boucle

Cycle dans un graphe

- C'est un chemin dont le début et la fin coïncident

Graphe acyclique

- Un graphe sans cycle

Arbre

- Un graphe acyclique connexe

- Un CSP dont le graphe est un arbre est "back-track free".

Ordre

- Est une relation d'ordre partiel binaire (≤) sur un ensemble S. Elle est réflexive, antisymétrique et transitive :

  - reflexive(S,≤) : $\forall X \in S,\ X \leq X$
  - anti symétrique(S,≤) : $\forall X,Y \in S,\ (X \leq Y\ \&\ Y \leq X \Rightarrow X=Y)$
  - transitive(S,≤) : $\forall X,Y,Z \in S,\ (X \leq Y\ \&\ Y \leq Z \Rightarrow X \leq Z)$

  Par exemple, "≤" sur les entiers naturels est un ordre partiel.

- L'ensemble S est *totalement* ordonné si tous ses couples d'éléments sont ordonnés : $\forall X,Y \in S$, $(X \leq Y \vee Y \leq X)$. L'ordre est alors appelé un *ordre total*.

  Par exemple, "≤" est un ordre total sur les entiers naturels.

NB : Annexe (suite)

Reprendre les fichiers « Suite-CLP-3A-05-06.pdf » et « Trans-crs-3-a-6-05.pdf »

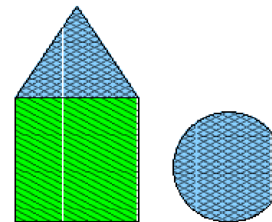
XVIII.B- Complément sur la CSP et CLP

XVIII.B.1- X dans CSP(X)

- Les contraintes sont exprimées sur des arbres (comme pour l'unification Prolog).
  Deux arbres (termes composés) seront identiques si leurs composants le sont.

- On peut manipuler des contraintes symboliques :

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



- Contraintes globales :

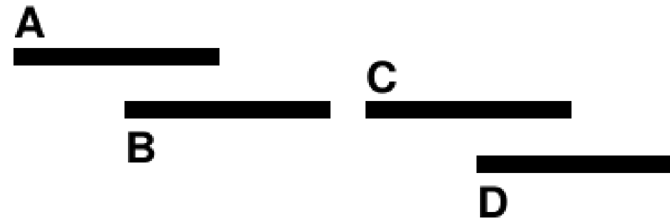
atmost(2, [X1 , X2 , X3 , X4 , X5], 1)

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

- Contraintes temporelles :



De :

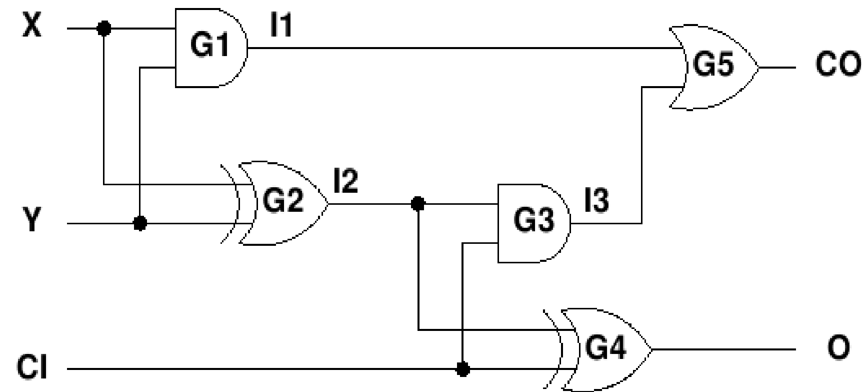
rel(AoverlapB, BbeforeC, R_AC)
& *rel(BbeforeC,CoverlapD, R_BD)*
& *rel(R_AC, CoverlapD, R_AD)*

On obtient :

R_AC= AbeforeC
& *R_BD=BbeforeD*
& *R_AD=AbeforeD*

Voir plus loin pour les détails des contraintes temporelles.

- Contraintes Booléennes : exemple additionner



Modélisation :

Variables : entrées/sorties des portes

Domaines : [0,1]

Contraintes Booléennes :

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

Ou encore :

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

XIX- Intérêts de la CSP (et de la CLP)

L'environnement CLP facilite la programmation déclarative
- meilleure spécification de l’intuition (l'interprétation attendue)
- prise en compte directe du domaine de discours

Le codage en UH n'est plus utile
- Les propriétés complexes du problème sont spécifiées d'une façon naturelle (e.g. contraintes arithmétiques, ensemblistes)
- Le problème lui même est représenté par un ensemble de règles.

Les contraintes sont utilisées :
- Pour représenter des relations entre objets,
- Pour calculer des valeurs basées sur ces relations (via les fonctions interprétées) :

  Z = 2Y +3 définit une relation entre Z et Y.
  Elle permet également de calculer la valeur de Z et de Y.

R**ègles récursives** : ajout dynamique de contraintes.

Contrôle de la résolution par les contrainte.
On vérifie à chaque pas de la résolution que le système est résolvable.

../..

- Utilisation des contraintes pour améliorer la "pureté" de Prolog :

Éviter dans certains cas l'utilisation de **cut** et de la **not**

p:- q, !, r.
p :- t.

Peut être remplacé dans certains cas par :
p :- q , r.
p :- q' , t.
q' : le complément de q (e.g. q dans {=,<}, q' dans{\=,>=}).

Idem pour **not**
cf. *pas_dans* au lieu de *not membre*, #\= au lieu de not =, ...

XIX.A- Paradigmes "Algorithmique" vs. "Contrainte"

• En programmation algorithmique conventionnelle :

- Le programme implante un algorithme particulier
- Le contrôle est explicite (logique de Hoar/séquentielle)
- L'exécution est "orienté-programme" (traitement)
- Les fonctions transforment un état de la mémoire vers un autre
- Le modèle de base d'exécution est séquentiel
- Le dévérminage et l'optimisation sont locaux

• En programmation par contraintes (CP)

- Le programme = une recherche dans un espace combinatoire
- Aucun algorithme spécifique employé, seulement des heuristiques
- L'exécution est "orientée-contraintes" (données)
- Les contraintes maintiennent l'état de calcul
- Le modèle d'exécution est parallèle et concurrent
- Le dévérminage et l'optimisation sont plutôt globaux
- Le raisonnement a lieu sur des réseaux de contraintes

XIX.B- Classes de problèmes (variés) résolus en CSP

- Traitement non déterministe dans les problèmes combinatoires.

- Programmation (logique, objet) en général et CSP en particulier

- Programmes relationnels (et **réversibles**)
  - Modélisation et opérations financières,...

- Analyse Numérique et Recherche Opérationnelle :
  Exprimer un grand nombre de contraintes =, ≠ , >, ... pour la résolution d'équations différentielles :
  - Approximation de Laplace,
  - Résolution Runge-Kutta, polynôme interpolateur de Lagrange
  - Applications en Électronique, Physique,...

- IA : Satisfaction de contraintes booléennes dans les S.E.
  Aide à la décision, logique temporelle

- Étiquetage d'arbres et les puzzles logiques

- TLN : traitement des nombres et des chaînes.

../..

- Contraintes géométriques (Infographie), IHM, Reconnaissance de formes, d'image et de scènes.

- Problèmes d'affectation/sélection sous contraintes
   - coloration de graphes
   - ordonnancement, planification, optimisation :
     PERT (e.g. Construction de ponts), bin Packing, Job Shop, Emploi du temps

- Applications d'ingénieurs :
   - Spécification de circuits électroniques :
     Description de comportement, Diagnostic et détection de pannes, Analyse et Simplification, ...
     Tout le domaine de la recherche Opérationnelle

- ....

XIX.C- Quelques Domaines d'application

Gestion du temps : gestion d’agenda, d’emploi du temps, planification d’équipes, de rotation d’équipes.

Gestion et affectation de ressources : gestion du personnel, de moyens de transports, gestion de production.

Planification et ordonnancement : planification de production, de livraison, de maintenance, d’interventions, d’itinéraires, ordonnancement d’ateliers.

Optimisation : optimisation de production, de moyens, d’investissements, de placements financiers, de coûts.

Gestion de la cohérence d’information : interfaces graphiques, édition électronique, tableurs, BDs, ...

Modélisation : modèles mixtes numériques/symboliques, géométriques, économiques, financiers.

Simulation comportementale de systèmes : Le système à étudier et son comportement sont définis par des relations (contraintes) entre les variables, y compris les entrées et les sorties. On peut automatiquement déduire les sorties à partir des entrées (et inversement); e.g. circuits électroniques, asservissement...

Vérification de spécification : le système de résolution de contraintes peut vérifier si le modèle de l’appareil ou du système est conforme à certaines propriétés, spécifiées par des contraintes (e.g. circuits électroniques)...

Diagnostic de pannes : Le système à étudier et son comportement fonctionnel sont définis par des contraintes. Un comportement anormal est mise en évidence lors de la vérification des contraintes...

XIX.C.1- Quelques Catégories d'applications

France en particulier

Industrie et production :

- Planification de la production (Elf, Kawasaki, Nippon Steel, Framatome).
- Configuration de clés hautes sécurité (Vachette).
- Aide à la conception de produits équilibrés - vitamine A - (Rhône-Poulenc).
- Planification de livraison de pièces détachées (Renault).
- Configuration de véhicules industriels (Renault).
- Gestion de stocks (Alcatel Câbles).
- Conception de composants chimiques (Sanofi).
- Planification de la maintenance et gestion de rendez-vous (Vitro Selenia, London Electricity).
- Optimisation de la découpe de bois (Cuisines Delacroix).
- Remplissage de cales de bateaux (Lyon)
- Conception et architecture des éléments de cuisines
- Scheduling : Usine de glace (Turkey), Planning d'entrepôts (Datatronics – France)

Transport :

- Planification de locomotives et gestion de personnel pour les trains (France, Australie, Malaisie, Pays-bas).
- Planification de bus (RATP, Espagne).
- Placement de robots de soudure (PSA).
- Planification de roulement d'équipage (Servair, Air France, Nynex)
- Gestion des bagages ou de portes d'embarquement aux aéroports (Allemagne, Singapour).
- Ordonnancement des équipes sur les chaînes de montages (Car Sequencing).
- Ordonnancement de construction de ponts.

....

Télécommunication :

- Optimisation de routage de réseaux de Télécommunication publics (CNET).
- Câblage d'immeuble (France Télécom).
- Management de personnel RFO
- Configuration automatique (ConBacon – Allemagne)

Militaire:

- Optimisation de plans de fréquence
- Planification de carrière
- Affectation de régions opérationnelles
- Planification d'escadrons mobiles (Gendarmerie Nationale).
- Lignes d'assemblage, conception, planning (Dassault Aviation)

Électronique :

- Placement et routage de VLSI (Bull, Siemens).
- Détection de pannes

Santé :

- Planification des rendez-vous médicaux, affectation de personnel (Hôpital de Strasbourg).
- Planning infirmerie (Paris)
- Reconstruction d'image en Tomographie (Allemagne)

Banque et finance :

- Analyse financière (projet IBM OTAS : Option Trading Analysis).

Éducation :

- Planification des examens (Université d'Angers).

....

XIX.D- 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.**

XIX.D.1- Objectifs

- 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, ../..

XIX.E- Langages et environnements spécifiques de CP

Langages proposant des outils de résolution CSP

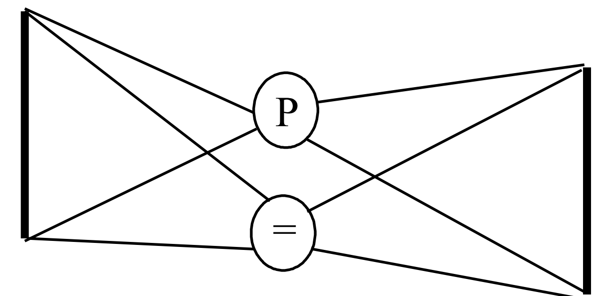
Approche traditionnelle en recherche opérationnelle (RO).
Analyse et développement des algorithmes dans un langage classique (impératif).
Techniques pré définies en RO (exemple : PERT)

XIX.E.1- Environnements de programmation spécifiques

Steele (Labo IA de MIT)
Basé sur la propagation locale
Explication du raisonnement, Application à TMS

SketchPad (MIT)
Application en dialogue homme-machine
Expressions de contraintes sur des objets graphiques primitifs (point, ligne, arc)

les deux segments doivent être parallèles (P)
et de même longueur (=)



XIX.E.2- Environnements génériques de programmation par contraintes

• Programmation logiques + contraintes (CLP)
• Programmation objet + contraintes (COP)

• Avantages et inconvénients des deux approches

• La CSP et les langages Hôtes :
  • **Impératif** (C)
  • **Fonctionnel** (LISP)
  • **Logique** (CLPR, PrologIII/IV, CHIP, Eclipse,Gnu Prolog)
  • **Objet** (C++, Smalltalk, IlogSolver,....)

• Environnements de Programmation par Contraintes

- Dans un langage dédié (C, Lisp)
  C/C++, Smalltalk ==> COP
  Prolog ==> CLP

- Dans un langage classique (C, Lisp, Java) + bibliothèques

XIX.E.3- Environnements CLP

CLP(R)

Traitement incrémental des contraintes par les méthodes linéaires (eg. Simplex)

PrologIII/IV

Idem CLP(R) + traitement d'arbres (infinis), listes, tuples, bool.
Utilise la méthode **Simplex** pour les numériques.

CHIP

Domaines finis (numérique, booléen)
Algorithmes implantant des techniques simples d'anticipation (Forward-checking) + choix de variable (principe First-fail)

PECOS et CHARME

Manipulation de contraintes symboliques et numériques.
Charme utilise la syntaxe de C et définit les tableaux
PECOS a une syntaxe de LISP.

XIX.E.4- Environnements COP

IlogSolver

A base de C++
Bibliothèque de résolution de contraintes

ThingLab

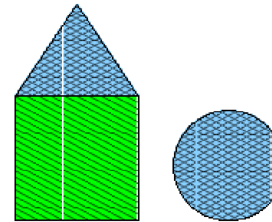
A base de Smalltalk + Bibliothèque de résolution de contraintes

XIX.F- X dans CSP(X)

- Les contraintes sont exprimées sur des arbres (comme pour l'unification Prolog).
  Deux arbres (termes composés) seront identiques si leurs composants le sont.

- On peut manipuler des contraintes symboliques :

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



- Contraintes globales :

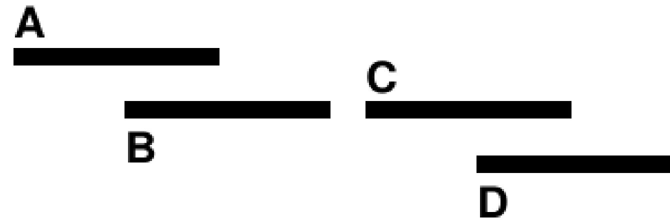
atmost(2, [X1 , X2 , X3 , X4 , X5], 1)

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

- Contraintes temporelles :



De :

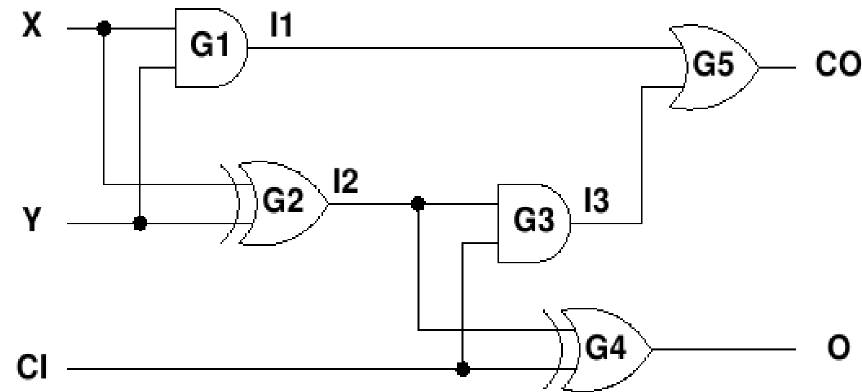
rel(AoverlapB, BbeforeC, R_AC)
& *rel(BbeforeC,CoverlapD, R_BD)*
& *rel(R_AC, CoverlapD, R_AD)*

On obtient :

R_AC= AbeforeC
& *R_BD=BbeforeD*
& *R_AD=AbeforeD*

Voir plus loin pour les détails des contraintes temporelles.

- Contraintes Booléennes : exemple additionner



Modélisation :

Variables : entrées/sorties des portes
Domaines : [0,1]
Contraintes Booléennes :

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

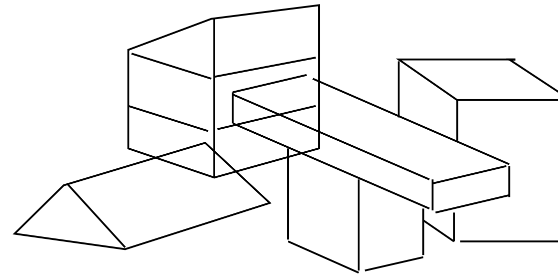
Ou encore :

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

XIX.G- Problème de reconnaissance de Scènes (Scene-labeling)

But : Identifier un objet 3D par son dessin 2D, définir ses contours et le désigner parmi d'autres objets dans une scène.

Exemple de scène :

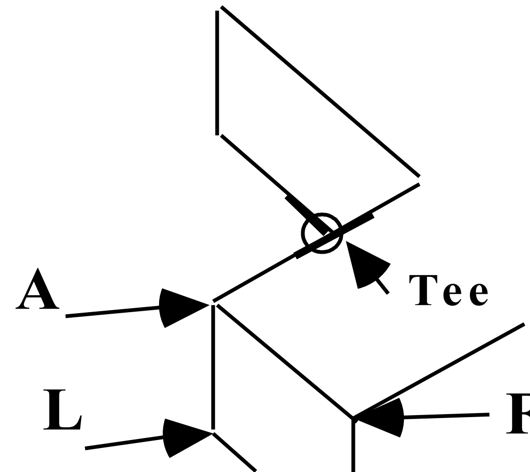


Éléments d'information :
Les catégories de points suivants sont affectées à l'intersection de 3 faces. On ne s'occupe pas des coins invisibles, c'est le but des calculs de les trouver :

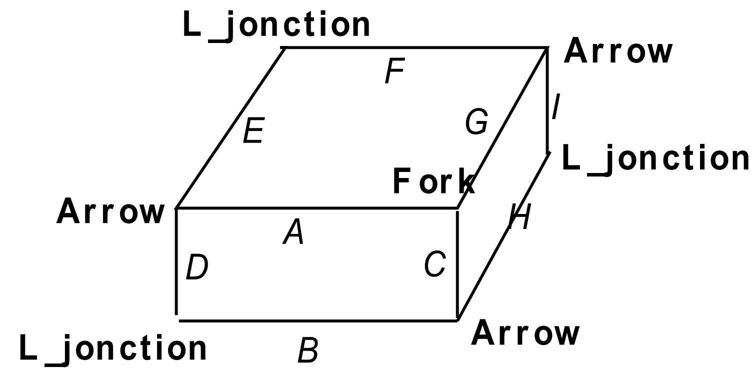
L_jonction : Une face visible et 2 cachées. Les deux segments visibles forment un **L** dans le coin.
Fork : Les 3 faces visibles, 3 segments visibles
Arrow : 2 faces visibles, une cachée, 3 segments visibles
Tee : 2 faces visibles, une cachée, 2 segments visibles dont un coupé par l'autre. Un Tee n'est pas une intersection de 3 faces.

C'est ce que l'on voit en 2D

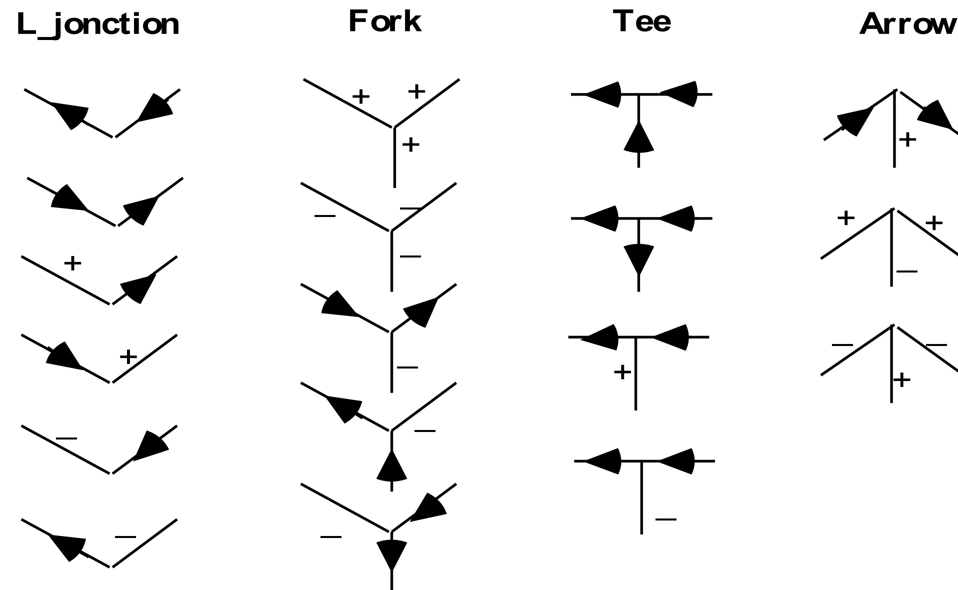
Exemple :



Exemple de dessin et d'application des catégories :



Les différent configurations possibles des 4 catégories ci-dessus :

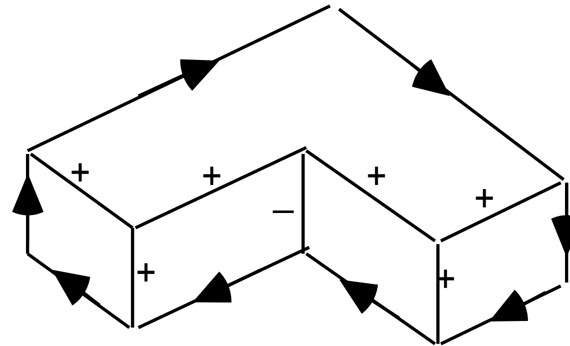


Interprétation des résultats :

Tout segment sera étiqueté par l'un des symboles **{→, ← , +, - }**.

- Les segments étiquetés par **{→, ←}** sont des segments aux *bords* d'objet. La *direction* d'une flèches est telle que si on avance dans cette direction, la face non cachée se trouve à notre droite.
- Un segment étiqueté par "**+**" est un segment *convexe* : ses deux faces s'éloignent de l'observateur.
- Un segment étiqueté par "**-**" est un segment *concave* : ses deux faces approchent l'observateur. Une des faces peut être cachée. C'est en fait le cas complémentaire des trois autres.

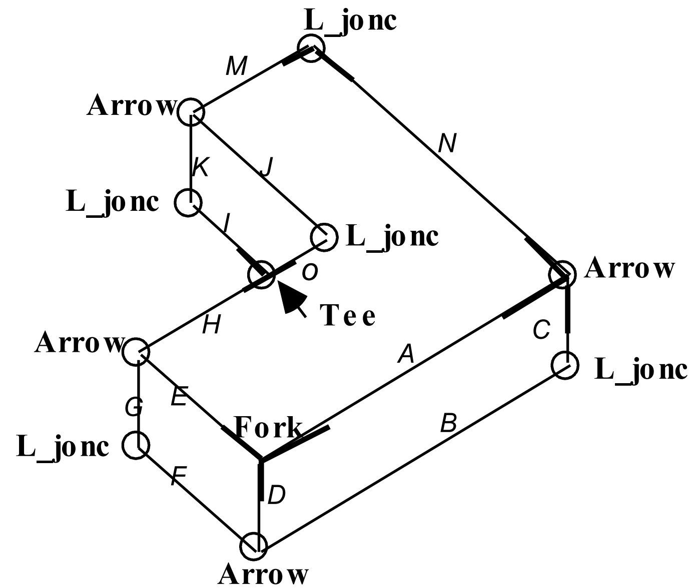
Exemple de résultat :



- Le **but** est de trouver les labels de chaque segment (→, ←, +, -) étant donné les contraintes imposées par les (catégories des) points d'intersections.

En détectant le label de chaque segments, on peut distinguer (et séparer) les objets dans l'espace, en particulier séparer les objets les uns des autres dans une scène.
Ainsi on interprète les dessins 2D sur papier par des vrais objets.

Exemple 1 : (Les lettres “A”..”O” sont des variables segments)



Variables : {A,..,O}

Domaines : {→, ←, +, -}

Contraintes :les liens entre les variables selon les 4 catégories.

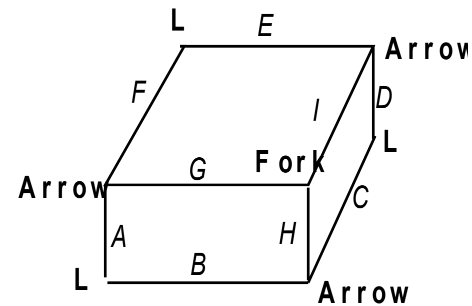
Programmation (indicatif, en PrologIII) :
- Chacune des 4 catégories a différentes configurations.
- La combinatoire peut être importante.
- Pour couvrir tous les cas possibles, on utilise les contraintes pour décrire les 4 catégories en réduisant la combinatoire.

Par exemple, pour représenter **arrow** :

```
arrow(A,B,C) ->
    only(2,<A,B,C>,'-')
    membre(A,['-','+'])  membre(B,['-','+'])  membre(C,['-','+']);

arrow(A,B,C) ->
    only(2,<A,B,C>,'+')
    membre(A,['-','+'])  membre(B,['-','+'])  membre(C,['-','+']);

arrow(A,B,C) ->
    only(2,<A,B,C>,'+')
    membre(A,['-','+'])  membre(B,['-','+'])  membre(C,['-','+']);

fork(A,B,C) ->
    alldif(<A,B,C>)
    membre(A,['<','>','-']) membre(B,['<','>','-']) membre(C,['<','>','-']);
```

```
fork('-','-','-') ->;
fork('+','+','+') ->;
.......

membre(X,Y) ->              % Contrainte d'appartenance.
     ...;                   % Y sera le domaine de X

only(A,B,C) ->              % contrainte symbolique de haut niveau
     atleast(A,B,C)         % traduit en d'autres contraintes .....
     atmost(A,B,C);
.....
```

Exemple : description du cube :

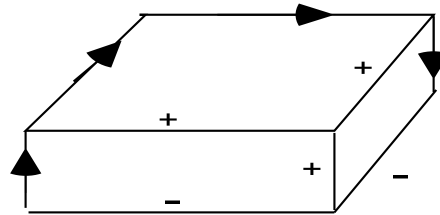



```
label([A,B,C,D,E,F,G,H,I]) ->
     arrow(F,A,G) fork(G,I,H)
     arrow(E,I,D)  arrow(B,C,H)
     ljonction(A,B)    ljonction(E,F)    ljonction(C,D);
```

Exemples d'interrogation :

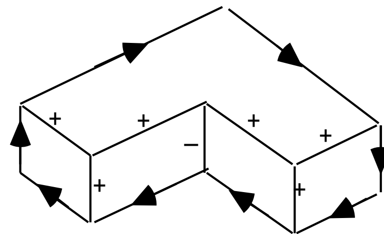
```
?- label([A,B,C,D,E,F,G,H,I]);
{A = '>', B = '<', C = '>', D = '<', E = '>', F = '<', G = '+', H = '+', I = '+'}
{A = '>', B = '-', C = '-', D = '<', E = '>', F = '<', G = '+', H = '+', I = '+'}
{A = '>', B = '+', C = '-', D = '<', E = '>', F = '<', G = '+', H = '+', I = '+'}
.......
```

Par exemple, la solution numéro 2 ci-dessus donne :



- Pour l'exemple ci-dessus, on aura parmi les solutions :

```
?- label([A,B,C,D,E,F,G,H,I,...,O]);
```



XIX.H- Raisonnement Temporel

Un **évènement** :
- est un intervalle de temps dont la durée est nulle.;
- correspond à une “pointe” de temps.

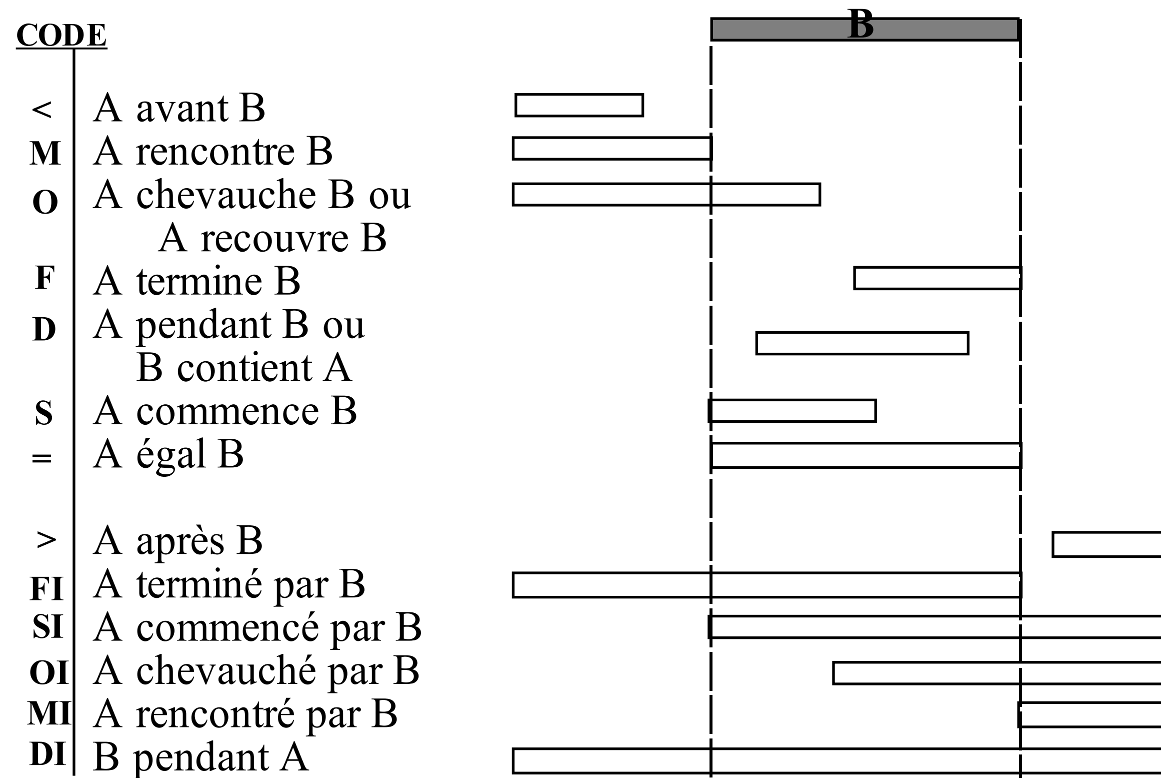
Un intervalle de temps :
- est associé à un processus qui a une durée;
- exprime 2 évènements **début** et **fin** d’un processus (fini)

- 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*.

L'algèbre d'intervalles : 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 événements. Pour n événements, 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.
- ......

Exemple (ordonnancement) :

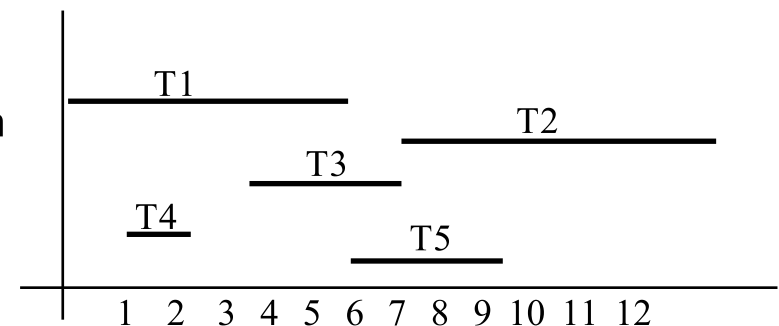
Soit un ensemble de tâches T1 .. T5 avec des relations temporelles et leur durées :

- *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 ==>



Exemple de codage (indicatif) :

```
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). ....
```

XIX.I- Exercice : Énigme policière (relations entre événements)

A : le propriétaire enclenche le système de sécurité
B : le système est armé (laisse 1 minute pour sortir)
C : le voleur est dans le magasin (3 minutes pour rentrer)
D : l'alarme sonne (un évènement)
E : la police est avertie (1 minute, 4 minutes pour arriver)
F : la police est sur place (évènement)
G : le voleur quitte le magasin (2 minutes pour s'échapper)
H : le coffre fort est ouvert (6 minutes pour l'ouvrir)

But : Sachant que

- le voleur n'était pas dans le magasin ni lorsque le propriétaire a armé l'alarme, ni lorsque la police est arrivée;
- la police a trouvé le coffre ouvert;

Ordonner les évènement et trouver les relations entre celles-ci.

(e.g. combien de temps après le déclenchement de l'alarme les sirènes des alarmes ont retentis ? Le voleur était seul ?...)

XIX.J- D'autres exemples d'utilisation de CSP

XIX.J.1- Planning & Scheduling

cf. l'exemple *car sequencing.*
Voir aussi les Tps.

XIX.J.2- 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.

XIX.J.3- 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.

XIX.K- 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:

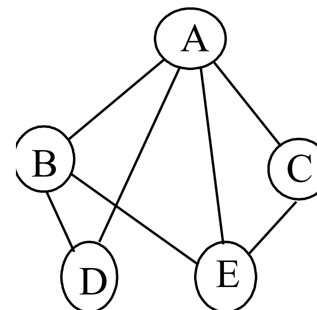
- Tout noeud de V2 correspond à un nœud distinct dans V1;
- Pour les noeuds x1,y1 dans V1 et x2,y2 dans V2 tels que x2,y2 correspond à x1,y1, si (x2,y2) est une arête de E2 alors (x1,y1) est une arête de E1.

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

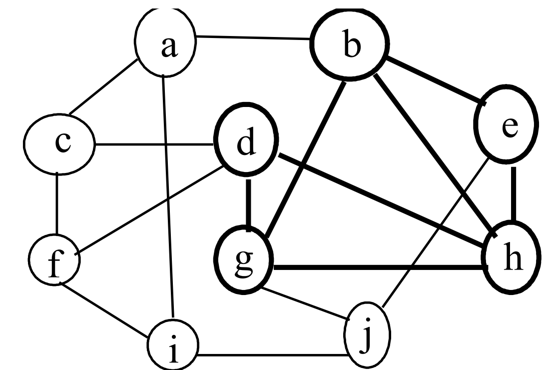
- Les variables : {A,B,C,D,E},
- Les domaines {a,b,...,i}.

- 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.



graphe G1



graphe G2

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

XIX.L- Solution au Problème (réel) de construction d’un Pont

On s'intéresse au problème d'ordonnancement disjonctif pour la construction d'un pont, défini par les contraintes

- de précédences
- de partage de ressources (exclusion mutuelle)
- de démarrage au plus tôt et au plus tard suivantes:

```
jobs([pa,a1,a2,a3,a4,a5,a6,p1,p2,ue,s1,s2,s3,s4,s5,s6,b1,b2,b3,b4,b5,
      b6,ab1, ab2,ab3,ab4,ab5,ab6,m1,m2,m3,m4,m5,m6,l,t1,t2,t3,t4,t5,ua, v1,v2,pe]).

Durations([0,4,2,2,2,2,5,20,13,10,8,4,4,4,4,10,1,1,1,1,1,1,1,1,1,1,1,1,16,
           8, 8,8,8,20,2,12,12,12,12,12,10,15,10,0]).
precedences([
        [pa,p1],  [pa,p2],  [pa,a1],  [pa,a2],  [pa,a5],  [pa,a6],
        [pa,ue],[pa,l],[p1,a3],[p2,a4],[a1,s1],[a2,s2],
        [a3,s3],[a4,s4], [a5,s5], [a6,s6], [s1,b1],
        [s2,b2], [s3,b3], [s4,b4], [s5,b5], [s6,b6],
        [b1,ab1], [b2,ab2], [b3,ab3], [b4,ab4], [b5,ab5],
        [b6,ab6], [ab1,m1], [ab2,m2], [ab3,m3], [ab4,m4],
        [ab5,m5], [ab6,m6], [m1,t1], [m2,t1], [m2,t2],
        [m3,t2], [m3,t3], [m4,t3], [m4,t4], [m5,t4],
        [m5,t5], [m6,t5], [l,t1], [l,t2], [l,t3], [l,t4],
        [l,t5], [ue,ua], [ua,pe], [m1,v1], [t1,v1], [m6,v2],
        [t5,v2], [v1,pe], [t1,pe], [t2,pe], [t3,pe],
        [t4,pe], [t5,pe], [v2,pe]
        ]).
```

```
ressources([
      [pile_driver,[p1,p2]],
      [crane,[l,t1,t2,t3,t5,t4]],
      [bricklaying,[m1,m2,m3,m4,m5,m6]],
      [carpentary,[s1,s2,s3,s4,s5,s6]],
      [excavator,[a1,a2,a3,a4,a5,a6]],
      [caterpillar,[v1,v2]],
      [concrete_mixer,[b1,b2,b3,b4,b5,b6]]
      ]).

/*
supdis(4,e(b1),e(s1)) is 4>=b1 + db1 - (s1 + ds1)
infdis(e(b1),e(s1),3) is b1 + db1 - (s1 + ds1) >=3
*/
distances([
      supdis(4,e(b1),e(s1)),
      supdis(4,e(b2),e(s2)),
      supdis(4,e(b3),e(s3)),

      supdis(4,e(b4),e(s4)),
      supdis(4,e(b5),e(s5)),
      supdis(4,e(b6),e(s6)),
      supdis(3,s(s1),e(a1)),
      supdis(3,s(s2),e(a2)),
supdis(3,s(s3),e(a3)),
supdis(3,s(s4),e(a4)),
supdis(3,s(s5),e(a5)),
```

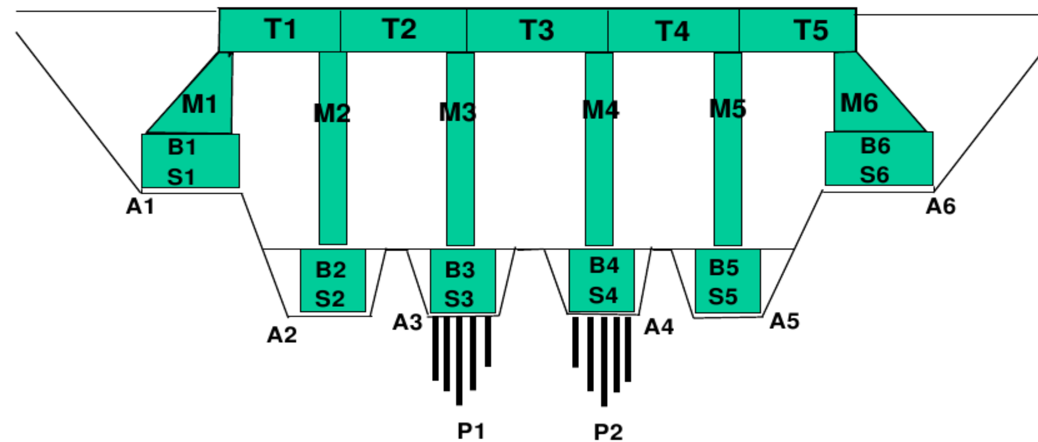
```
supdis(3,s(s6),e(a6)),
supdis(3,s(s3),e(p1)),
supdis(3,s(s4),e(p2)),
infdis(s(s1),s(ue),6),
infdis(s(s2),s(ue),6),
infdis(s(s3),s(ue),6),
infdis(s(s4),s(ue),6),
infdis(s(s5),s(ue),6),
infdis(s(s6),s(ue),6),
infdis(s(ua),e(m1),-2),
infdis(s(ua),e(m2),-2),
infdis(s(ua),e(m3),-2),
infdis(s(ua),e(m4),-2),
infdis(s(ua),e(m5),-2),
infdis(s(ua),e(m6),-2),
supdis(30,s(l),s(pa)),
infdis(s(l),s(pa),30),
supdis(0,s(ab1),e(b1)),
supdis(0,s(ab2),e(b2)),
supdis(0,s(ab3),e(b3)),
supdis(0,s(ab4),e(b4)),
supdis(0,s(ab5),e(b5)),
supdis(0,s(ab6),e(b6)),
infdis(s(ab1),e(b1),0),
infdis(s(ab2),e(b2),0),
infdis(s(ab3),e(b3),0),
infdis(s(ab4),e(b4),0),
infdis(s(ab5),e(b5),0),
```

```
infdis(s(ab6),e(b6),0)
]).
```

a) En utilisant le prédicat Prolog **nth(N, List, Elem)** qui succède si Elem est le Nieme élément de la liste List, écrire une prédicat **duree(Tache, Duree)** pour récupérer la durée d'une tâche.

b) Écrire une requête GNU-Prolog pour trouver la date d'achèvement optimale des travaux.

XIX.L.1- Dessin du problème 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	-

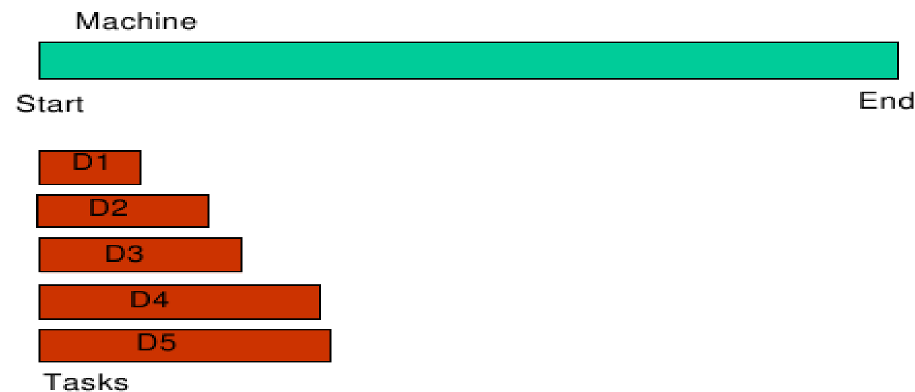
Les contraintes temporelles :

- 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).
```

XIX.M- Exemple Bin Packing

<A ajouter.> Donné en couts « Stratégie de résolution de problèmes »

XX- Le modèle concurrent pour l'exécution de CSP (domaine fini)

- Chaque contrainte est un agent actif en attente d'un *message*;
- Les variables représentent les canaux de communications
- La modification d'une variable est un message qui déclenche toutes les contraintes connectées à cette variable ;
- Chaque contrainte déclenchée se réveille, renforce la consistance du réseau puis se re-suspend;
- Après un nombre fini de mis à jour, le réseau atteint un point fixe et se stabilise (état de consistance globale)
- Il n'y a aucune différence spécifique entre la résolution et la propagation de contraintes;
- La résolution a lieu par les phases de réduction et recherche;
- La phase de recherche est en général une énumération des valeurs;
- La recherche est en général **NP ou exponentielle**
- On utilise des techniques de réduction et de recherche concurrentes

XXI- La complexité des problèmes CSP (domaine fini)

XXI.A- Le problème des N-reines

- Les contraintes à satisfaire

une solution pour N=8 :

	A	B	C	D	E	F	G	H
1	♛							
2							♛	
3					♛			
4								♛
5		♛						
6				♛				
7						♛		
8			♛					

- 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 $|D|^{|V|}$

XXII- Éléments théoriques du CSP

Quelques éléments formels (simplifiés) du paradigme contraintes.

XXIII- définitions et terminologies

- Notions de *variable*, *domaine*, *assignation (instanciation), contraintes, satisfaction, solution,* ...

- Variables **V** = $\{X_1,...,X_n\}$ à valeur dans les domaines **D** = $\{D_1...D_n\}$

- Les principaux domaines :
  - Entiers (domaine fini, infini) : {1..5} ou {1..infini}
  - Booléens (domaine fini) : {0,1} = {vrai, faux}
  - Réels (infini, intervalle), Rationnels
  - Symboliques (énumératifs) : {lu,ma,me,je..., di}

- Un **label** est une paire (variable $\times$ valeur) : <x,v>

- Un **label composé** est un tuple de paires : $(<x_1, v_1>, \ldots, <x_n, v_n>)$

- C : Un ensemble de contraintes (fonction économique)

../..

- Une *contrainte* $\mathbf{c_j(X_{i1},...,X_{ik})}$ entre k variables de S est un sous-ensemble de $D_{i1} \times ... \times D_{ik}$ spécifiant les valeurs des variables mutuellement compatibles.

- Une contrainte désigne conceptuelle ment un ensemble de labels composés légaux.

Exemple : pour $X,Y \in \{1..5\}$, la contrainte $C_{X,Y}$: “X+Y = 4” désigne l’ensemble :

{(<X,1>, <Y,3>), (<X,2>, <Y,2>), (<X,3>, <Y,1>)}.

- Une contrainte restreint l'ensemble des labels composés constructibles avec les variables et les domaines du problème .

XXIII.A- Notion de satisfaction

Une relation binaire entre un label (composé) et une contrainte :

- $S = \{x_1, ..., x_n\}$ l'ensemble fini de variables
- C_S les contraintes sur l'ensemble S
- *Variables*(C_S) = S
- Un ensemble de labels composés **L** avec ses variables X.
  L *satisfait* une contrainte C ssi X est élément de C :

satisfait$((<x_1, v_1>, ..., <x_k, v_k>) , C_{x1,...,xk})$
$\equiv (<x_1, v_1>, ..., <x_k, v_k>) \in C_{x1,...,xk}$

Plus généralement :**satisfait**$((<x, v>, C_x)) \equiv (<x, v>) \in C_x$

../..

Exemple :

Pour le problème de 4-reines, l'ensemble des contraintes désigné par $C_{A,B,C,D}$ exprime l'ensemble des labels composées :

{(<A,2>, <B,4>,<C,1>,<D,3>), (<A,3>, <B,1>,<C,4>,<D,2>)}.

	A	B	C	D
1			•	
2	•			
3				•
4		•		

	A	B	C	D
1		•		
2				•
3	•			
4			•	

Pour L=(<A,3>, <B,1>,<C,4>,<D,2>) , on a :

satisfait(L, $C_{A,B,C,D}$) puisque L$\in C_{A,B,C,D}$

Un problème de satisfaction de contraintes (CSP) est défini par le triplet **(V, D, C)**.

- Une *solution* à un CSP est une application $V \rightarrow D$ qui satisfait toutes les contraintes C simultanément produisant les tuples de labels composés (<x1, v1>, ..., <xn, vn>).

$\forall$ csp((V,D,C)) : $\forall x_1, ..., x_n \in V$: $\forall v_1 \in D_{X_1}, ..., v_n \in D_{X_n}$:
est_solution((<x1, v1>, ..., <xn, vn>), (V,D,C))
ssi
(($V=\{x_1, ..., x_n\}$) & $\forall c \in C$: **satisfait**((<x1, v1>, ..., <xn, vn>),c).

Un CSP est *satisfiable* si le tuple (<x1, v1>, ..., <xn, vn>) existe (est calculée).

- Selon le problème en cours, Le *but de la satisfaction* peut être de :
  -trouver *aucune* solution.
  -trouver *une* solution.
  -trouver *toutes* les solutions
  -trouver une ou toutes les solutions *optimales*.

Remarque :

On s’intéresse plus particulièrement aux CSP finis (*problèmes d’instanciations consistantes*).

L’étude des problèmes CSP où les variables peuvent avoir un domaine infini (cf. problème numériques) ou ceux où l’ensemble des variables peut dynamiquement changer (selon certaines valeurs, de nouvelles variables peuvent émerger dans le problème) nécessitent d’autres techniques spécialisées.

XXIII.B- Représentation des contraintes

- S = {x1, ..., xn} l'ensemble des variables
- $C_S = C_{x1,...,xn}$ les contraintes sur S
- Contraintes *unaire, binaires*, *n-aires*

Contraintes binaires : contraintes ne contenant que des contraintes binaires et unaires.

Exemple de contrainte unaire : $x \in \{1,2,3\}$
Exemple de contrainte binaire : $x + y < 10$
Exemple de contrainte 3-aires : $x + y * z \neq 23$

1) On peut représenter une contrainte (numérique) sous la forme d'un système d'équations-inéquations (e.g. X+Y > 10).

Les contraintes symboliques peuvent être traduites en numériques :
voisin(A,B) :- A=B±1

2) On peut représenter une contrainte sous la forme d'une fonction appliquant chaque tuple de label (<x1, v1>, ..., <xn, vn>) dans {vrai, faux}.

Une contrainte peut être représentée comme l'ensemble des labels composés légaux sur les variables du problème.

Exemple : ../..

Exemple : Avec X ∈ {1,2,3}, Y∈ {4,5,6,7} et la contrainte “X+Y < 8”, cet ensemble sera :

{(<X,1 >, <Y, 4>), (<X, 1 >, <Y, 5>), (<X,1 >, <Y, 6>),
(<X,2 >, <Y, 4>), (<X, 2 >, <Y, 5>), (<X,3 >, <Y, 4>)}

- Nous prendrons cette représentation logique.
- Nous l’utiliserons dans le problème de réduction (suppression des éléments redondants de cet ensemble).

3) Une contrainte binaire peut être représentée par une matrice.

Exemple :
X ∈ {1,2,3}, Y ∈ {4,5,6,7} et la contrainte “ X + Y est impaire”

C_{xy}

X\Y	**4**	**5**	**6**	**7**
1	V	F	V	F
2	F	V	F	V
3	V	F	V	F

Ce qui représente les tuples (labels composés)
{(<x,1>, <y,4>), (<x,1>, <y,6>), ..., (<x,3>, <y,6>)}

- Une grande partie des problèmes CSP s’exprime à l’aide des contraintes binaires (et unaires).
- Tout CSP n-aire peut être transformé en un CSP binaire.

XXIII.C- Transformation n-aire vers binaire

Soit $x_1,...,x_k$ des variable et C une contrainte k-aire sur elles.

On définit une variable W (dont le domaine est l'ensemble de tous les labels composés de C) et k contraintes binaires.

Chacune de ces contraintes connecte W à une des variables $x_1,...,x_k$.

La contrainte binaire entre W et une variable x_i est une **projection** de W en x_i :

project(V_W, (<X, V_X>))

i.e. X doit avoir la même valeur que W en X. V_W et V_X sont les valeurs prises par W et X.

Une projection de (<x,1>, <y,1>, <z,2>) en x est <x,1>, i.e. x est contraint à prendre la valeur 1 par cette projection.

../..

Exemple de transformation :

- Soit x,y,z $\in$ {1,2} et la contrainte C : **"x,y et z ne prennent pas la même valeur".**

- Les 6 tuples légales pour cette contrainte ternaire seront :

A= {(<x,1>, <y,1>, <z,2>), (<x,1>, <y,2>, <z,1>),
(<x,1>, <y,2>, <z,2>) .... , (<x,2>, <y,1>, <z,1>)}

- On définit W dont le domaine est exactement l'ensemble A et 3 contraintes binaires du type
$project(V_W, (<x, V_X>))$.

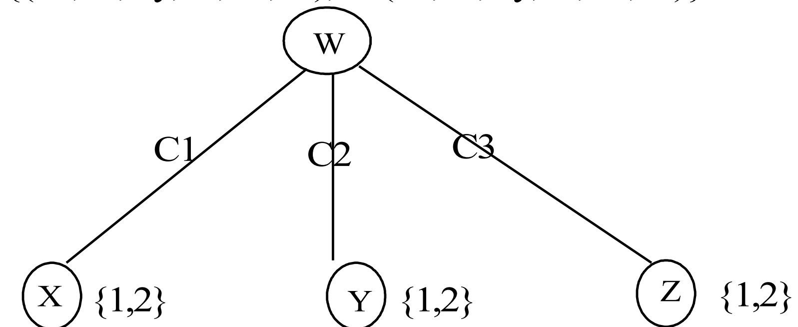
Par cette contrainte, si W prend la valeur (<x,1>, <y,1>, <z,2>) alors x doit prendre la valeur 1.

De même pour y et z.

transformation contrainte ternaire en binaire

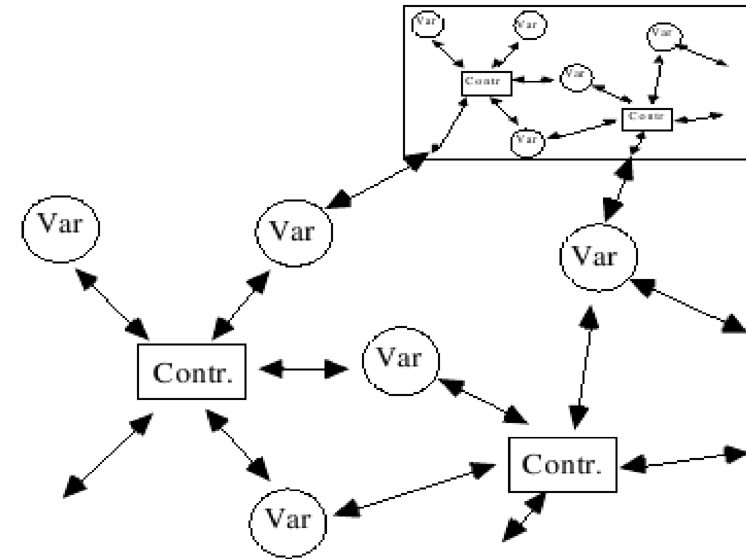
La nouvelle variable W et son domaine :

{(<x,1>, <y,1>, <z,2>), ... (<x,2>, <y,1>, <z,1>)}



Les contraintes binaires Ci imposent que le label en X (Y et Z) soit une projection de la même valeur de W

XXIII.D- Propagation des contraintes



En explicitant au maximum un système de contraintes (réseau minimal), la recherche de solutions devient plus efficace.
Par inférence (utilisé sur demande car nécessitant un nombre exponentiel de nouvelles contraintes), la recherche produit une solution (si elle existe) directement en évitant tout échec.

Dans un réseau de contraintes avec k variables, un algorithme de *k-consistance* garantit que toute instanciation consistante de *k-1* variables peut être étendue à n´importe quelle kème variable.

Les algorithmes de k-consistance sont basés sur le principe que si une valeur ne peut participer à la construction d´une solution d´un sous-réseau de k variables alors cette valeur ne peut pas, non plus, rentrer dans la construction d´une solution complète.

Point important : toute contrainte non binaire est transformable en un ensemble de contraintes binaires (voir document du cours). Ce principe permet de raisonner uniquement sur les contraintes binaires.

Pour un réseau (graphe) de contraintes, il existe 3 types de consistance : nœud, arc et chemin.

XXIII.D.1- Consistance de Noeud (Node Consistancy)

C'est la forme la plus simple de consistance.
Un nœud représenté par une variable V est nœud-consistant (dans le graphe des contraintes) si toutes les valeurs du domaine de V vérifient les contraintes unaires (dont le nombre de variables =1 comme dans *X > 1*) sur V.
Les valeurs du domaine qui ne vérifient pas ces contraintes sont éliminées du domaine de V.

XXIII.D.2- Consistance d'Arcs (Arc consistancy)

Une fois la consistance de nœud vérifiée, on considère les contraintes binaires faisant participer 2 variables (qui correspondent aux arcs) sur des paires de variables.

La consistance d´arcs d´un réseau est obtenue en vérifiant la consistance sur toutes les paires de variables X,Y (liées par un ensemble de contraintes binaires $C_{x,y}$).

Cette procédure (connue sous le nom REVISE) vérifie et élimine, des domaines de X et Y, les valeurs incompatibles avec les contraintes $C_{x,y}$.
On note qu'il y a une direction sur les contraintes et la consistance de $C_{x,y}$ ne vaut pas forcément pour $C_{y,x}$. On note également que pour une 3e variable Z du réseau des contraintes, lorsque la consistance des contraintes $C_{y,z}$ auront été vérifiées (modifiant les domaines D_y et D_z), on doit revoir la consistance de $C_{x,y}$ car le domaine D_y a pu changer.
On notera que suite à la consistance d'arcs, le réseau de contraintes restant peut ne pas avoir de solution.

Exemple : si le domaine des variables X et Y est réduit à une seule valeur, en présence de la contrainte $X \neq Y$, on peut conclure immédiatement sur un échec.

Par contre, dans X,Y,Z $\in$ {1,2} avec la contrainte $X \neq Y$, $Y \neq Z$, $X \neq Y$.

la vérification de la consistance d'arcs ne peut supprimer aucune valeur de ces domaines. Pourtant, le réseau restant n'a pas de solution (d'où l'intérêt de la *consistance de chemin*, voir ci-dessous). De ce fait, la consistance d'arc ne suffit pas à supprimer le besoin de retours arrières (car échec local est possible).

N.B. : On dira que qu'un solveur qui se contente de ces vérifications sans le mécanisme de retour arrière est incomplet (ne donne pas toujours les bonnes réponses).
Par contre, muni du mécanisme de retour arrière, ce solveur devient complet.

La complexité de la procédure REVISE est $O(e.k^2)$ où **k** borne la taille des domaines de valeurs des variables et **e** le nombre de contraintes binaires.
La complexité spatiale de cet algorithme est la même que sa complexité temporelle.
La valeur k^2 est le nombre maximal de relations binaires dans le réseau C.

La consistance **partielle** d'arc (par exemple X#=Y pour l'égalité en Gprolog) est une version simplifiée de l'algorithme REVISE qui modifie seulement les bornes des domaines.

La consistance totale (Full Arc Consistancy, par exemple X#=#Y pour l'égalité en Gprolog) considère la totalité du domaine de chaque variable.
La consistance totale procède à davantage de propagations que la version partielle.

XXIII.D.3- Consistance de Chemin (Path consistancy) ou la K-consistance

Comme on l'a constaté ci-dessus, la consistance d'arc ne supprime pas toutes les valeurs incompatibles et ne suffit pas à supprimer les retours arrières dans le solveur.

On peut étendre l'algorithme de consistance d'arc à la consistance du chemin (plusieurs variables en lien via des contraintes, cf. **X#\=Y, X#\=Z, Z #\= Y** en GProlog).

Un graphe de contraintes est dit *k-consistant* si ce qui suit est vrai :

- soit la consistance vérifiée pour k-1 variables du graphe.
- S'il reste une valeur correcte pour la kème variable compatible avec toutes les contraintes du graphe, alors le graphe est dit *k-consistant*.

La consistance de noeud est 1-consistant, la consistance d'arc est 2-consistant.

La complexité de l´algorithme *de consistance de chemin* est $O(n^3.k^3)$.

Exemple : si le domaine D={1,2} pour X,Y,Z avec les contraintes X#\=Y, X#\=Z, Z #\= Y (2 à deux différentes) pour les variables, alors l'algorithme conclue sur un échec immédiat.

Étant donné le coût important des algorithmes de consistance de chemin, on se contente en général d'une vérification de 3-consistance. Un autre algorithme appelé *restricted path-consistancy* existe également (équivalent à une consistance d'arc non directionnelle).

Important : si un graphe de contraintes sur N variables est N-consistant (path-consistant), alors aucune recherche (aucun retour arrière) n'est nécessaire pour trouver une solution.

Par contre, même si le graphe est K-consistant pour K<N, alors on pourrait avoir besoin de procéder à des retours arrières.

Cependant, la consistance d'arcs suivie d'énumération (après l'énumération pour une variable, le réseau de contraintes élimine les valeurs incompatibles des domaines des autres) permettent de minimiser le nombre de retours arrières (qui sont à leur tour implantés dans une version optimisée dite *intelligent back-tracking*).

XXIII.E- Caractéristiques de la CLP

- Spécification d'un programme ≡ expression des contraintes

- En CLP, les contraintes sont utilisées pour spécifier les entrées et les sorties des programmes :

*Contraintes en entrées => **Programme** => Contraintes en sortie*

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

- Le langage est déclaratif
  => les contraintes non directionnelles

- L'utilisation de CLP facilite la programmation déclarative grâce à une meilleure spécification de l'intuition.

- On peut travailler directement dans le domaine du problème (e.g. R) sans devoir coder les termes par ceux de U_H.

- **Le solveur de contraintes contrôle la résolution**.
  Il doit pouvoir déterminer à chaque étape si une collection de contraintes est solvable (consistant) ou non.

- Dans un programme CLP :
  - On spécifie les propriétés (complexes) du problème d'une façon naturelle par des contraintes.

- Le problème lui même est représenté par un ensemble de règles.

Exemple :

```
circuit_parallele (V, I, R1, R2) ->
resistance (V, I1, R1)
resistance (V, I2, R2)
{I1 + I2 = I};

resistance(V, I, R) -> {V = I * R};
```

- Les règles récursives permettent d'ajouter dynamiquement des contraintes.

```
Chiffres_differents(<>) ->;
Chiffres_differents(<x>.s) ->
 Hors_de(x, s)
 Chiffres_differents(s),
 {0 <= x <= 9};

Hors_de(x, <>) ->;
Hors_de(x, <y>.s) -> Hors_de(x, s), {x#y};
```

- Le solveur incrémental de CLP vérifie la consistance du système de contraintes à toute étape de la résolution. (Simplex, Gauss, ... incrémental)