

Ecole Centrale de Lyon

Optionnel 2A

Algorithmes et Raisonnement

Programmation Logique
avec Contraintes

BE 4

A. Saidi, E. Dellandréa
Département M. I.
2016-2017

Table des matières

I Prise en main des contraintes (BProlog)	4
I.1 Le principe de résolution CLP	4
I.2 Quelques prédictats BProlog utilisant des contraintes	5
II Des exemples simples	6
II.1 Un premier exemple	6
II.2 Valeurs d'équation simple	7
II.3 Têtes et des pattes : une équation simple	7
II.4 Calcul de la factorielle (réversible)	8
II.5 La découpe	9
II.5.a Optimisation (stratégie B&B et minof)	10
II.5.b Remarques importantes sur minof / maxof	12
II.6 Exemple SENDMORY	13
II.7 Exemple N-reines	14
II.7.a Une solution numérique	15
II.7.b Une solution booléenne	15
II.8 Remarques sur les contraintes	17
III Exercices (avec solution) sur les contraintes numériques	19
III.1 Rappel sur le schéma CLP	19
III.2 L'exemple des pierres (vu en cours)	19
III.3 Tous_diff	20
III.4 Puzzle numérique	21
III.5 Exemple de multiplication	22
III.5.a Le code Prolog	22
III.6 Coloration	24
III.6.a Le code	24
III.6.b Complément sur l'énumération (dans coloration)	25
III.6.c Une autre stratégie indépendante d'une énumération particulière	25
III.6.d Encore une autre stratégie	25
III.6.e Une 3e stratégie	26
III.7 Notes sur le calcul des extrema	27
III.7.a Un exemple	28
IV Exercices à commencer en séance	29
IV.1 Problème de transport (2)	29
IV.2 Problème de localisation d'entrepôts (2)	30
IV.3 Ordonnancement PERT (2)	31
IV.4 Ordonnancement disjonctif (2)	32
IV.5 JobShop avec ressources et sans bénéfice (2)	32
IV.6 JobShop avec bénéfice (3)	33

IV.7 Problème d'emploi du temps (2)	33
IV.8 Réunion (3)	34
IV.9 Planification (5)	35
IV.10 Qui a Gagné la Médaille d'Or ? (3)	36

I Prise en main des contraintes (BProlog)

Pour modéliser un problème (logique) avec contraintes, et même si ce modèle est trivial, on définit un schéma contraintes (basique) en définissant le triplet **(V, D,C)** :

- 1) les Variables (*V*) et leurs domaines (*D*)
- 2) les contraintes et relations sur ces variables (*C*)
- 3) au besoin, générer des valeurs pour les variables (énumération)

Au besoin, on définit des contraintes supplémentaires éventuelles de vérification des résultats (selon le problème traité).

I.1 Le principe de résolution CLP

Lorsqu'un programme CLP est interrogé par Prolog (muni d'un moteur de résolution de la logique des prédicats étendu aux contraintes), les réponses sont obtenues, après toutes les simplifications et optimisations, de la manière suivante. Un ensemble de contraintes est **consistant** si le domaine (de valeurs) d'aucune de ses variables n'est vide.

1- Ou bien l'ensemble des contraintes est inconsistante : il y a échec.

Exemple-1 (les contraintes sur les entiers commencent par le symbole '#')

?- X #> Y , X #< Y. % contradictoire!
→ Non (pas de réponse)

On peut également donner une définition du domaine de X,Y (la réponse ne change pas)

?- [X,Y] :: 1..2, X #> Y , X #< Y. % X,Y dans 1..2
→ Non (pas de réponse)

Le symbole '%' commence un commentaire dans les programmes.

2- Ou bien l'ensemble des contraintes est consistante et mène à une réponse unique :

Exemple-2 :
?- [X,Y] :: 1..2, X #> Y.
→ X = 2 , Y = 1

Ici, après simplification, les domaines de X et de Y sont réduits à un singleton chacun.

3- Ou bien l'ensemble des contraintes est consistante et mène, après simplifications, à plusieurs solutions. Ce que l'on appelle une **contrainte-réponse**.

Exemple-3 :
go(X,Y) :- [X,Y] :: 1..4, X #> Y+1 .
→ ?- go(X,Y). → X = 3..4, Y = 1..2

Une contrainte réponse représente l'état (domaines) des variables après toutes les simplifications faites par le moteur de Prolog.

→ En général, si le domaine d'une variable n'est pas réduit à une seule valeur, on procèdera par une énumération de valeurs pour obtenir les différents résultats.

Par exemple :

```
go(X,Y) :-  
    [X,Y] :: 1..4,    % ou "[X,Y] in 1..4"  
    X #> Y+1,  
    labeling([X,Y]).
```

→ ?- go(X,Y).

X = 3, Y = 1

X = 4, Y = 1

X = 4, Y = 2

Remarque : quand on avait obtenu ci-dessus $X = 3..4$, $Y = 1..2$

l'ensemble de réponses peut faire croire qu'il y aurait 4 réponses possibles (dont $X=3$, $Y=2$). Or, un test permet de voir que $X=3$, $Y=2$ n'est pas une réponse valide.

Pourquoi ?

Lorsque les contraintes ne "vident" pas les domaines des variables (comme dans $[X,Y] :: 1..4$, $X \#> Y$, $Y \#> X$), Prolog présente la contrainte-réponse avec les domaines de X et de Y où pour chaque valeur de X, il reste au moins une valeur pour Y.

Ce qui ne veut pas dire : pour chaque valeur de X, chaque valeur du domaine de Y constitue une réponse (et vice versa).

Pour connaître les véritables réponses, on procède par une énumération (par *labeling*).

I.2 Quelques prédictats BProlog utilisant des contraintes

Ces éléments sont utilisés dans les premiers exemples suivants. Les autres éléments nécessaires seront rappelés au fur et à mesure.

Prédicats/Contraintes	Signification	Exemples
Var :: inf..sup	Var ∈ inf .. sup	$X :: 1..0$
$[V_1,..,V_n] :: inf..sup$	$V_1,..,V_n \in inf .. sup$	$[X,Y,Z] :: 1..10$
Expr $\#=$ Expr	Équation dans Z	$X + 1 \#= Y * 2 + Z.$
Expr $\#\neq$ Expr	Inéquation dans Z	$X + 1 \#\neq Y * 2 + Z.$
$\#<$, $\#>$, $\#= <$,	D'autres opérateurs	$X \#< Y.$
labeling([V1,...,Vn])	Énumération de valeurs pour V1 ... Vn	$[X,Y,Z] :: 1..10,$ $X \#< Y, Z \#= 2*X + Y,$ $labeling([X,Y]).$

Voir plus loin pour une liste plus détaillée des contraintes.

Voir aussi la documentation BProlog en ligne. (vous l'avez dans le répertoire de Bprolog).

Dans la suite de ce document, on étudie quelques exemples puis on traite des exercices.

II Des exemples simples

Créer un fichier contenant les petits programmes suivants ; le charger sous BProlog puis poser des questions.

II.1 Un premier exemple

a) Le tout premier exemple d'une équation linéaire (on peut laisser Z sans domaine) :

exemple(X, Y , Z) :-

X in 5..20, Y :: [2,4,6,8,12, 15], %Z n'est pas limitée, ce sera un entier relatif
X #< Y, Z #= 2*X -Y.

Signification (au sens équationnel) :

$X \in 5..20, Y \in \{2, 4, 6, 8, 12, 15\}, X < Y, Z = 2X - Y$.

On pose la question % ne taper pas l'invite " ?- "

?- **exemple(A,B,C).**

→ réponse : $X \in 5..14, Y \in \{6, 8, 12, 15\} Z \in -5..22$

On constate une réduction des domaines (voir explications page suivante).

b) On pose la même question avec une énumération d'une des variables (par *labeling*) :

?- **exemple(A,B,C), labeling([B]).** % noter bien que *labeling* demande une liste, ici [B]

→ $X = 5, Y = 6, Z = 4 ?;$ avec ';' , on demande une autre solution
 → $X \in [5..7], Y = 8, Z \in [2, 4, 6] ?;$
 → $X \in [5..11], Y = 12, Z \in [-2, 0, 2, 4, 6, 8, 10] ?$
 → ...

c) Avec plus d'énumération :

?- **exemple(A,B,C), labeling([A,B]).** % on pourrait énumérer 'C' aussi.

→ $A = 5, B = 6, C = 4 ;$
 → $A = 5, B = 8, C = 2 ;$
 → ...

Remarque : *labeling* aurait pu être placé dans le prédicat exemple même.

A retenir :

- Lisez toujours les messages de BProlog après le chargement de votre code !
Corrigez les erreurs (et warnings) signalées : elles s'avèrent souvent importantes.
- Le schéma contraindre-puis-générer d'un programme avec contraintes.
- *Var :: min..max* : définition du domaine de Var (dans Z). Idem *Var in min..max*
- *[Liste_de_Vars] :: min..max* : définition du domaine de plusieurs variables.
- *X #= Y, X #< Y, X #\= Y...* : contraintes d'égalité, ordre, différence,...
- *labeling([Var])* : énumération des valeurs du domaine d'une variable Var
- *labeling([Liste_de_Vars])* : énumérations de valeurs d'une liste de variables

II.2 Valeurs d'équation simple

Trouver les valeurs X et Y de telle sorte que les contraintes suivantes soient satisfaites :

- X et Y peuvent prendre des valeurs allant de 1 à 5,
- $X < Y$,
- $X^2 + Y^2 = 25$

Indication : poser les contraintes puis énumérer.

Solution : on en profite pour rappeler les éléments de base de la CSP (le triplet V,D,C).

1- Posons la question sans préciser les domaines (ne pas taper le prompt "|?–") :

?- $X \#< Y, X*X + Y*Y \#= 25$. % X, Y dans \mathbb{Z}

→ X et Y : -268435455..268435455

2- Posons la même question avec plus de contraintes

?- $X \#< Y, X*X + Y*Y \#= 25, X \#> 0, Y \#> 0$.

→ $X = 1..11, Y = 2..12$

On constate le travail plus approfondi de la réduction des domaines.

3- Posons maintenant des questions avec les domaines :

?- $[X, Y] : :1..5, X \#< Y, X*X + Y*Y \#= 25$.

→ $X : 1..4, Y = 2..5$

4- Et enfin avec énumération sur X :

?- $[X, Y] : :1..5, X \#< Y, X*X + Y*Y \#= 25, labeling(X)$.

→ $X=3, Y=4$

Notons que seule l'énumération sur X suffit et les dépendances font le reste !

II.3 Têtes et des pattes : une équation simple

Une équation simple pour calculer le nombre de têtes et des pattes de L lapins et de P pigeons (une variable commence par une majuscule, un prédicat par une minuscule)

nombre_tetes_pattes(Lapins, Pigeons, Tetes, Pattes) :-

Tetes $\#=$ Lapins + Pigeons,

Pattes $\#=$ 2 * Pigeons + 4 * Lapins.

Questions :

?- nombre_tetes_pattes(6,2,U,V).

→ $U=8, V=20$

?- $X \#> 0, Y \#> 0$, nombre_tetes_pattes(X,Y,8,20).

→ $X = 6, Y = 2$

N.B. l'exemple est trop simple mais on peut bien distinguer le triplet de définition (V, D, C) vu en cours.

II.4 Calcul de la factorielle (réversible)

`fact(0, 1). % la "factorielle de 0 =1" est vraie.`

`fact(N, M) :-`

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

Questions :

`fact(4,X).` → $X=24$

`fact(X, 24), !.` % le cut est nécessaire pour éviter une boucle infinie

→ $X=4$

La présence du *cut* (qui veut dire : *donner une seule solution*) est nécessaire et l'on se contente de la première réponse.

Sinon, Prolog tentera de trouver une autre solution (en explorant toutes les valeurs des entiers : 0 .. 268435455 pour lesquelles la factorielle = 24). Ce qui nécessite beaucoup de ressources.

☞ Comparer avec la version du Prolog standard (suivante) de factorielle et montrer pourquoi la version Prolog pure n'est pas réversible.

`fact(0,1).`

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

Et poser les mêmes questions.

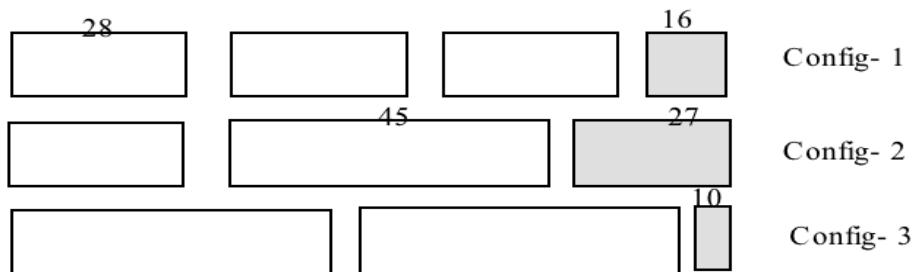
A retenir :

- Prolog est un langage relationnel : *les prédictats ne peuvent renvoyer autre valeur que true ou false (succès ou échec)*.
- Donc, pour dire " factorielle de 0 = 1 ", on n'écrit pas en Prolog
`fact(0) :- 1 .`
Mais `fact(0, 1).`
- Pour faire des calculs, il faut utiliser des variables en paramètre qui reçoivent une valeur *si le prédictat lui-même est vrai* (représente la valeur true).
- Chaque attribution de valeur à une variable active des réflexes qui permettent de (re) vérifier les consistances (du noeud et d'arc) des contraintes mettant en jeu cette variable.
- Les tests de Prolog standard tels que $X>0$ exigent que X ait une valeur au moment du test. Par contre, la version "contrainte" ($X \#> 0$) impose que X soit (désormais) plus grand que 0.
- Avec les contraintes, les tests sont retardés jusqu'à l'attribution d'une valeur à chaque variable.
- Du fait du caractère équationnel des écritures, les propriétés relationnelles et réversibles des programmes sont davantage préservées (vs. Prolog standard).
- On utilise *cut* (noté par '!') si l'on veut se contenter de la première solution.

II.5 La découpe

Une machine sait découper de barres de 28 et 45 dans une barre d'un mètre.
Pour satisfaire une commande de 36 barres de 28 cm et 24 barres de 45 cm, combien de ces barres sont elles nécessaires ? Optimiser ce nombre et les chutes.

- a) Vous pouvez dans un premier temps observer le travail de simplification des domaines sans procéder à une énumération.



- b) Puis, appliquer l'énumération par **labeling(Variables)**.
c) Ensuite, on demandera la minimisation (*voir 2 pages plus loin pour un exemple*).

Etude de la solution :

Etape-1 : sans énumération (et sans optimisation des Chutes)

Le triplet CSP :

$$\begin{aligned} V &= \{\text{Config_1}, \text{Config_2}, \text{Config_3}\} \cup \{\text{Chutes}\} \\ D_{\{\text{Config_1}, \text{Config_2}, \text{Config_3}\}} &= 0..36, \quad D_{\text{Chutes}} = \mathbb{N} \\ C &= \text{voir le code.} \end{aligned}$$

Pour illustrer le fonctionnement, on n'utilise pas *l'énumération* dans cette version.

`decoupe1(Chutes, Config_1, Config_2, Config_3) :-`

```
[Config_1, Config_2, Config_3] :: 0..36
, Chutes #= 16 * Config_1 + 27 * Config_2 + 10 * Config_3
, 3*Config_1 + Config_2 #= 36
, Config_2 + 2*Config_3 #= 24.
```

Questions :

`?- decoupe1(Chutes, Conf1, Conf2, Conf3).`

après simplification des contraintes par BProlog, on obtient :

→ Chutes → 64..960, Conf1 → {4,6,8,10,12},
Conf2 → {0,6,12,18,24} Conf3 → {0,3,6,9,12}

Cette réduction des domaines ne suffit pas à proposer une solution nette et chiffrée.

Remarque : relâcher les contraintes et remplacer dans le code

`3*Config_1 + Config_2 #= 36, Config_2 + 2*Config_3 #= 24`
par : `3*Config_1 + Config_2 #>= 36, Config_2 + 2*Config_3 #>= 24`
et observez les différences.

Etape-2 : on ajoute la génération des valeurs par *labeling(liste_de_variables)* :

```
decoupe2(Chutes, Config_1, Config_2, Config_3) :-
```

```
[Config_1, Config_2, Config_3] :: 0..36
, Chutes #= 16 * Config_1 + 27 * Config_2 + 10 * Config_3
, 3* Config_1 + Config_2 #= 36
, Config_2 + 2* Config_3 #= 24
, labeling([Config_1, Config_2, Config_3]).
```

Posons une question ...

```
?- decoupe2(Chutes, Conf1, Conf2, Conf3) .
```

```
→ Chutes = 712  Conf1 = 4  Conf2 = 24  Conf3 = 0
→ Chutes = 612  Conf1 = 6  Conf2 = 18  Conf3 = 3
→ Chutes = 512  Conf1 = 8  Conf2 = 12  Conf3 = 6
→ Chutes = 412  Conf1 = 10  Conf2 = 6  Conf3 = 9
→ Chutes = 312  Conf1 = 12  Conf2 = 0  Conf3 = 12
```

II.5.a Optimisation (stratégie B&B et minof)

On peut demander la minimisation de la valeur d'une variable par :

minof(Prédicat, Variable_a_minimiser)

→ demande à minimiser *Variable_a_minimiser* dans *Prédicat*

Etape 3 : on demande à minimiser les **Chutes**.

Pour ce faire, la meilleure solution est de demander la génération des valeurs sous la contrainte d'optimisation (par *minof*) :

minof(énumération, Variable_a_minimiser) demande à minimiser *Variable_a_minimiser* pendant l'énumération.

```
decoupe3(Chutes, Config_1, Config_2, Config_3) :-
```

```
[Config_1, Config_2, Config_3] :: 0..36,
, Chutes #= 16*Config_1 + 27*Config_2+10* Config_3
, 3* Config_1 + Config_2 #= 36
, Config_2 + 2* Config_3 #= 24
,minof(labeling([Config_1, Config_2, Config_3]), Chutes).
```

```
?- decoupe3(Chutes, Conf1, Conf2, Conf3) .
```

ptab → Chutes = 312 Conf1 = 12 Conf2 = 0 Conf3 = 12

N.B. : **Chutes** ne figure pas dans l'énumération mais est liée aux autres variables par les contraintes.

A retenir :

- *minof(Prédicat, Variable_A_minimiser)*

Si Prédicat est une énumération (labeling), la contrainte de minimisation est prise en compte pendant l'énumération.

- *La variable à minimiser n'a pas besoin de figurer explicitement : elle doit néanmoins être liée aux autres variables par des contraintes.*

- *Symétriquement : maxof.*

Remarques sur l'exemple découpe :

Reformulons cet exemple dans la version suivante où le nombre des pièces à livrer (36 et 24) est considérer comme un *minimum* : remarquer les deux opérateurs $\#>=$:

decoupe4(Chutes, X,Y,Z) :-

```
[X,Y,Z] :: 0..36
, Chutes #>= 16*X+27*Y+10*Z
, 3*X + Y #>= 36
, Y + 2*Z #>= 24
, labeling([X,Y,Z]).
```

Questions : (il y a plusieurs réponses possibles) :

?- decoupe4(Chutes, X,Y,Z).

- Chutes = 972, X = 0, Y = 36, Z = 0
- Chutes = 982, X = 0, Y = 36, Z = 1
- Chutes = 992, X = 0, Y = 36, Z = 2 ...

Utilisons minof dans la requête :

?- Chutes #>0, minof(decoupe4(Chutes, X,Y,Z), Chutes).

- Chutes = 312, X = 12, Y = 0, Z = 12

Ici, on a besoin de borner **Chutes** (essayer sans).

Si on veut éviter de borner **Chutes**, on peut insérer *minof* dans le prédicat decoup4 :

decoupe4_bis(Chutes, X,Y,Z) :-

```
[X,Y,Z] :: 0..36
, Chutes #>= 16*X+27*Y+10*Z
, 3*X + Y #>= 36
, Y + 2*Z #>= 24
, minof(labeling([X,Y,Z]),Chutes).
```

?- decoupe4_bis(Chutes, X,Y,Z), Chutes).

- Chutes = 312, X = 12, Y = 0, Z = 12

Quelle différence entre decoupe4 et decoupe4_bis ? :

- Dans **decoupe4** : l'énumération (par *labeling*) a lieu puis, lors de poser la question, on demande à minimiser la solution. Ce qui est un schéma générer-tester.
 - Dans **decoupe4_bis** : la minimisation est réclamée pendant l'énumération.
- Cette seconde version est bien plus efficace.

II.5.b Remarques importantes sur *minof* / *maxof*

D'une manière générale, on peut argumenter qu'en présence de maxof /minof, l'énumération (par *labeling*) n'est pas nécessaire puisque l'on s'intéresse aux minima / maxima des domaines.

La remarque est (en général) juste mais selon la formulation des contraintes, ces minima peuvent ne pas fournir une réponse.

Dans l'exemple de la découpe :

`decoupe5(X,Y,Z,Chutes) :-`

```
[X,Y,Z :: 0..36
, Chutes #= 16*X+27*Y+10*Z
, 3*X + Y #>= 36
, Y + 2*Z #>= 24.]
```

Avec : `?- decoupe5(X, Y, Z, Chutes).`

→ Chutes : 0..1908, X, Y, Z : 0..36

On note bien qu'après les simplifications, le minimum du domaine de la variable **Chutes** = 0.

Posons une question en demandant la minimisation :

`?- minof(decoupe5(X, Y, Z, Chutes), Chutes).`

→ variables pas suffisamment instanciées.

☞ De plus, le min de *Chutes* = 0, mais avec cette valeur, pas de solution.

Par contre, la requête suivante donnera les bonnes réponses (2 requêtes)

`?- decoupe5(X, Y, Z, Chutes), minof(labeling(Chutes), Chutes).`

→ bonne réponse.

grâce à *labeling* qui trouve les combinaisons de valeurs consistantes.

A retenir : maximisation / minimisation

- avec *minof* / *maxof* : `minof(Goal, Exp).`

- lors d'utilisation de *labeling* sur une liste de variables :

`labeling([mimimize(Exp)], Liste).`

Ou `minof(labeling(Liste), Exp).`

Préférez, quand c'est possible, le second schéma.

II.6 Exemple SENDMORY

Trouver des valeurs (0..9) pour les variables {S,E,N,D,M,O,R,Y} telles que l'addition soit juste. On demande à ce que S et M soient non nulles.

$$\begin{array}{r} \text{S E N D} \\ + \\ \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$

Spécification CSP = triplet (V, D, C)

V = { S,E,N,D,M,O,R,Y }

D = 0..9

C= {S et M soient différents de nul

Tous les chiffres deux-à-deux différents

La somme soit juste}

Version 1 : vérifier l'addition en multipliant les chiffres par leur poids (unité, dizaine, centaine, ...).

send(Liste) :-

Liste=[S,E,N,D,M,O,R,Y],

alldifferent(Liste), % tous les éléments 2 à 2 différents

Liste :: 0..9, % domaine des variables

[S,M] :: 1..9,

1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E

#=10000*M+1000*O+100*N+10*E+Y,

labeling(Liste).

Solution : [9,5,6,7,1,0,8,2]

N.B. : Sans l'énumération par *labeling*, une première réponse partielle (une *contrainte-réponse*) est :

?- send(L). % L=[S,E,N,D,M,O,R,Y]

→ S=9, E=2..8, N=2..8, D=2..8,

M=1, O=0, R=2..8 , Y=2..8

Version 2 : poser l'addition en termes de chiffres et retenues.

Cette version réclamera des variables supplémentaires (les retenues Ci).

Principe : poser E + D #= Y + 10 * C1

C1 + N + R #= E + 10* C2

...

Version 3 : une autre manière de modéliser " tous différents " dans ce problème.

On peut considérer une fonction booléenne (0/1) de {S,E,N,D,M,O,R,Y} dans {0..9} dont la somme est =1 :

Soit f_x,d avec x → {S,E,N,D,M,O,R,Y} avec le domaine des variables dom = 0..9

qui peut prendre la valeur 0 ou 1, avec la contrainte $\sum_{d=0}^{d=9} f_{xd}=1$ pour x donné.

L'exemple N-reines suivant exploite (dans sa 2e solution) cette modélisation.

Remarque sur la complexité de ces 3 versions :

Une estimation rapide de la complexité est de $|D|^{|V|}$ où $|D|$ est la taille du domaine des variables et $|V|$ le nombre de variables.

Voir l'exemple N-reines suivant qui discute de la complexité.

A retenir :

- *alldifferent(Liste_de_vars)* : imposer aux variables à être 2-à-2 différentes.
- Le domaine d'une variable peut être plusieurs fois redéfinis (par réductions successives).
- Il y a souvent plusieurs méthodes de résolution pour un problème ; on choisira celle dont la complexité est minimale.
- Le prédicat d'énumération labeling énumère des valeurs pour les variables dans un ordre quelconque. Il se décline sous d'autres formes, en particulier *labelingff* qui permet d'énumérer d'abord les variables dont le domaine est le plus réduit (d'où le nom avec *ff* pour first-fail = celle qui risque le plus d'échouer). Un exemple est donné dans la page suivante. Il y a d'autres possibilités (voir la doc Bprolog).

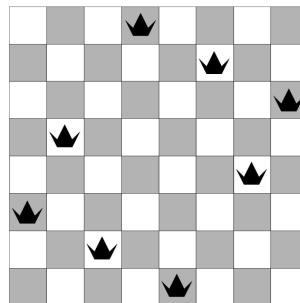
II.7 Exemple N-reines

Placer N pions (reines) sur un échiquier NxN).

Les contraintes : jamais 2 pions sur la même ligne/colonne/diagonale.

Un exemple avec N=8 et une de ses solutions (les colonnes : de A à H, les lignes de 1 à 8)

Choix possibles de la spécification des variables et leurs domaines (pour N=8)



1- Un ensemble de 8 *variables* $V=\{V1..V8\}$ chacune prenant une valeur dans l'intervalle $D=\{A..H\}$ ou dans l'intervalle $D=\{1..8\}$. \rightarrow combinatoire $= 8^8$

2- En numérotant les cases (1..64), un ensemble de 8 variables $V=\{V1..V8\}$ chacune prenant une valeur dans l'intervalle $D=\{1..64\}$. $\rightarrow 64^8 = 8^{16}$

3- un ensemble de 64 variables $V=\{V1..V64\}$ chacune prenant une valeur dans l'intervalle $D=\{0,1\}$.
 $\rightarrow 2^{64} > 8^{21}$

...

☞ Choix selon la complexité minimale de $|D|^{|V|}$

☞ Voir la solution

II.7.a Une solution numérique

Questions :

`reines(4, L). → L= [2,4,1,3]`

`reines(8, L). → L= [1,5,8,6,3,7,2,4]`

`reines(16, L). → L= [1,3,5,2,13,9,14,12,15,6,16,7,4,11,8,10]`

Le code :

`reines(N, L) :-`

`length(L,N), % générer une liste de N variables`

`L : 1..N,`

`safe(L),`

`labelingff(L). % labeling fonctionnera aussi`

% imposer la bonne position des pions

`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 (depuis 1)`

`safe(L). % vérifier (récursevement) les autres pions ensemble`

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

`noattack([Y|L],X,I) :- % I progresse de 1 à N (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.`

A retenir :

- *labeling(Liste_vars)* énumère les valeurs du domaine de chaque variable.

- *labelingff(Liste_vars)* fait de même mais en choisissant d'abord la variable dont le domaine (restant) est de taille minimum.

- Il existe d'autres options pour ce prédictat (voir la doc Bprolog) permettant une meilleure adaptation.

II.7.b Une solution booléenne

On considère N=4 et l'échiquier contiendra 16 booléens. Parmi ces 16 variables, seules 4 seront vraies dans la solution et donneront par leur position la solution.

La solution envisagée utilisera les prédictats prédéfinis de BProlog sur les variables booléennes. La solution sera représentée de la manière suivante :

On représente chaque cellule (ligne i, colonne j) par une booléenne X_{ij} .

Les contraintes seront :

- pour toutes les lignes L_i ($i : 1 .. 4$) : **une seule** des $B_{ij} = 1$ ($j=1..4$)

- pour toutes les colonnes C_j ($j : 1 .. 4$) : **une seule** des $B_{ij} = 1$ ($i=1..4$)

- pour les 10 diagonales ($N=4$) : **au plus une** des cases occupées.

Les prédictats portant sur les contraintes booléennes sont dérivés de ceux sur les entiers en réduisant leurs domaines à {0,1}.

Il s'agit des prédictats :

at_least_one([X, Y, Z, ..]) : au moins une des contraintes (ou variables sous contraintes) de la liste [X,Y,Z,..] est vraie (vaut 1).

Par exemple : ?- $X \#(A \#> B)$, $Y \#(A \#> C)$, $at_least_one([X, Y])$.

Donnera $X, Y : 0..1$ et $[A, B, C]$ dans Z .

→ Une seule des 2 contraintes $A \#> B$, $A \#> C$ doit être vraie.

at_most_one([X, Y, Z, ..]) : au plus une des variables de la liste [X,Y,Z,..] est vraie (vaut 1).

only_one([X, Y, Z, ..]) : seule une des contraintes de la liste [X,Y,Z,..] est vraie (vaut 1). Ce prédictat est une conjonction de *at_least_one* et *at_most_one*.

Exemple : l'opérateur Xor (cas "A xor B" = vrai) :

xor([A,B],1) :- only_one([A,B]).

Questions : ?- xor(L,1), labeling(L). → L=[0,1] ou L=[1,0]

A l'aide de ces prédictats, on choisit une formulation booléenne du problème Nreines pour N=4.

Solution indicative à base des booléennes pour N=4 (il est tout à fait possible de passer à la version pour N quelconque) :

On remarque que pour les lignes et les colonnes, on exige seulement une case occupée (only_one) mais pour les diagonales, au plus une case (at_most_one).

```
reines4([[X11,X12,X13,X14], %Ligne 1
[X21,X22,X23,X24], %Ligne 2
[X31,X32,X33,X34],
[X41,X42,X43,X44]]) :-
```

% Contraintes sur les Lignes

only_one([X11,X12,X13,X14]), % la première ligne

... % et les 3 autres lignes

% Contraintes sur les Colonnes

only_one([X11,X21,X31,X41]), % la première colonne

... % et les 3 autres colonnes

% Contraintes sur les Diagonales principales :

at_most_one([X11,X22,X33,X44]),

at_most_one([X14,X23,X32,X41]),

... % et les 8 autres diagonales (partielles)

% et on énumère les valeurs :

labelingff([X11, X12, ...,X44]).

?- reines4(L).

→ L = [[0,0,1,0],[1,0,0,0],[0,0,0,1],[0,1,0,0]]

et la solution symétrique L = [[0,1,0,0],[0,0,0,1],[1,0,0,0],[0,0,1,0]]

II.8 Remarques sur les contraintes

1- Les contraintes sur les entiers existent également pour les booléens. De plus, il existe un mécanisme de passage des contraintes sur les entiers vers les booléennes (appelé **réification**).

On peut par exemple associer une variable booléenne B à une contrainte numérique :

X : $:0..10$, (**X** $\#> 10$) $\#<=>$ **B**.

C'est à dire, pour X dans $0..10$, établir l'équivalence (logique) entre $X > 10$ et la valeur de B. La réponse à une telle requête est $B=0$ ($B=false$).

De même, les exemples de réification

(**X** $\#> 10$) $\#= B$ (X sans domaine précisé) ou (**X** $\#> 10$) $\#<=>$ **B**
donneront $B=0..1$, X dans \mathbb{Z} .

2- Plus généralement, les contraintes sur les booléennes sont définies à l'aide de la contrainte principale **count**.

count(Val, Liste, Op, N)

S'il y a **K** éléments de Liste dont la valeur = **Val**, alors on impose la contrainte :

$K \text{ } Op \text{ } N$. (par exemple, $K \#= N$) où **Op** est l'opérateur d'égalité

Op peut être n'importe quel opérateur de contrainte arithmétique.

Exemple (**count(1, L, #>, 5)** : plus de 5 variables dans L seront > 1)

?- $L=[A,B,C,D,E,F,G,H,I,J]$, $L : : 0..10$, **count(1, L, #>, 5)**, **labeling(L)**.

Soit K=le nombre des '1' dans L . On demande à ce que K $\#> 5$.

→ $L = [0,0,0,0,1,1,1,1,1]$

$L = [0,0,0,1,0,1,1,1,1] \dots$

Ainsi, **count** permet de définir le cardinal d'une liste respectant certaines contraintes.

Un autre exemple : Soit un problème (portant sur X et Y) formulé par :

X,Y in 0..3, tels que l'une seule des 3 conditions suivantes soient satisfaites :

X > Y+1, X-Y = 2, Y-X = 2

On peut utiliser count pour ne satisfaire qu'une seule des 3.

?- $[X,Y] \text{ in } 0..3$, $A1 \#<=> (X \#> Y+1)$, $A2 \#<=> (X-Y \#= 2)$, $A3 \#<=> (Y-X \#= 2)$,
count(1, [A1,A2,A3], #=, 1), **labeling([X,Y])**.

→ $X=0, Y=2 \quad X=1, Y=3 \quad X=3, Y=0$

Les autres contraintes (dites globales) :

atleast(N, Liste,V)

Au moins N éléments de Liste ont la valeur V.

atmost(N, Liste, V) idem

exactly(N, Liste, V). idem

global_cardinality(Liste, Vals) généralise ces contraintes (voir Doc BProlog).

Voir la doc BProlog pour toutes les contraintes spécifiques sur les booléennes.

Un exemple :

```
?- [X,Y] in 0..3, A1 #<=> (X #> Y+1), A2 #<=> (X-Y #= 2), A3 #<=> (Y-X#= 2),
    exactly(1,[A1,A2,A3], 1).
→ X,Y = 0..3, A1, A2,A3 = 0..1
```

Autres contraintes intéressantes :

cumulative, serialized, etc. (voir la doc BProlog)

III Exercices (avec solution) sur les contraintes numériques

III.1 Rappel sur le schéma CLP

Les étapes de spécification d'une solution :

- 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.

III.2 L'exemple des pierres (vu en cours)

Un épicer sait qu'il aura des marchandises pesant de 1,2,3 ... 40 kilos à vendre.

Il fait ses calculs et décide de se procurer seulement 4 poids.

Quel sont les poids qu'il aura choisi ?

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

Spécification (pseudo code) du prédictat principal (contenant l'énumération finale) :

pierres(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),

 labeling(L).

NB : l'ordre établi entre les P_i permet d'éviter des solutions symétriques.

Spécifier les relations :

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

tous_les_poids_mesurables(_, A, A). % on aura tout vérifié

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) :

B_i=0 : le poids participe pas ; **B_i=1** : poids sur la balance droite, **B_i=-1** : sur la balance gauche.

peser(A, [P1, P2, P3, P4]) :- % Vérifier que le poids A est measurable par les Pi

A #= B1*P1 + B2*P2 + B3*P3 + B4*P4,

[B1, B2, B3, B4] :: -1 .. 1. % -1, 0 ou 1

→ Solution : $L = [1, 3, 9, 27]$.

Remarques :

- La vérification des propriétés (une étape de la démarche de spécification avec des contraintes) est ici simple : le prédicat **tous_les_poids_mesurables** permet de faire cette vérification !

Pour un autre exemple de ce type de vérifications, voir le "**Problème de localisation d'entrepôts**" plus loin.

- Il n'y a pas ici d'optimisation particulière ici.

- La vérification des propriétés (une étape de la démarche de spécification avec des contraintes) est ici simple : le prédicat **tous_les_poids_mesurables** permet de faire cette vérification !

Pour un autre exemple de ce type de vérifications, voir le "**Problème de localisation d'entrepôts**" plus loin.

- Les prédicats permettent d'injecter les contraintes au fur et à mesure (chaque contrainte injectée doit être compatible avec les précédentes)

→ On peut proposer une solution directe sans passer par des prédicats :

Utiliser 4 variables $\in [-1, 0, 1]$ pour chaque poids allant de 1 à 40, puis imposer les contraintes :

$\forall i \in 1..40$

$$B_{1i}.P_1 + B_{2i}.P_2 + B_{3i}.P_3 + B_{4i}.P_4 = i$$

Cette solution sera strictement équivalente à la précédente mais sans passer par les prédicats.

III.3 Tous_diff

Ecrire un prédicat **tous_diff(Liste)** qui impose à ses éléments (des entiers) à être deux à deux différents. Tester avec une liste d'entiers.

Ensuite, définir 5 variables avec des valeurs dans le domaine 1..5, énumérer et tester votre solution.

N.B. : Ce prédicat a été utilisé plus haut (la version prédéfinie : *alldifferent / 1*)

Solution :

tous_diff([]).

tous_diff([X|L]) :- pas_dans(X,L) , tous_diff(L).

pas_dans(X,[]).

pas_dans(X,[Y|L]) :- X #\= Y , pas_dans(X,L).

Questions : **tous_diff([1, 2, 3]).** → true

tous_diff([1, 2, 3, 2]). → no

Test :/..

Test : on remarque bien le triplet (V,D,C) dans la requête suivante :

Créer une liste de 5 variables dans le domaine 1..5 avec des valeurs différentes.

```
length(L,5),      % création d'une liste de 5 variables
L :- 1..5,         % chaque variable ∈ 1..5
tous_diff(L),     % contraintes (autres que les domaines)
labeling(L).       % énumération
→ L=[1,2,3,4,5] ...
```

N.B. : Il y a le prédéfini *alldifferent* dans Bprolog (utilisé plus haut) qui fait la même chose.

III.4 Puzzle numérique

Soit une matrice de 9 cases (3 x 3) contenant des nombres de 1 à 9.

Trouver les nombres qui composent le carré de telle sorte que les contraintes suivantes soient vérifiées :

- La somme de chaque ligne, de chaque colonne et de chaque diagonale est égale à 15,
- Tous les nombres sont différents.

Exemple de solution :

```
[2,7,6]
[9,5,1]   Il y a 8 solutions.
[4,3,8]
```

Solution :

```
solution(L) :-
    L = [X11,X12,X13, X21, X22, X23, X31, X32, X33],
    L :- 1..9,
    alldifferent((L))           % ou tous_diff
    X11+X12+X13 #= 15, X21+X22+ X23 #= 15,      % lignes
    X31+X32+ X33 #=15,
    X11+X22+X33 #= 15, X13+X22+ X31 #= 15,      % Diagonales
    X11+X21+ X31 #=15, X12+X22+ X32 #=15,      % colonnes
    X13+X23+ X33 #=15,
    labeling(L).
```

Question :

`solution(L) → L = [2,7,6,9,5,1,4,3,8]`

```
X11 = 2 , X12 = 7, X13 = 6
X21 = 9 , X22 = 5, X23 = 1
X31 = 4, X32 = 3, X33 = 8 ....
```

III.5 Exemple de multiplication

Trouver des valeurs (0,..9) pour chaque variable X_i tels que la multiplication suivante soit juste. Chaque chiffre ne doit apparaître que **deux fois** exactement.

N.B. : Dans un premier temps, ne tenez pas compte de la dernière contrainte ci-dessus. Ensuite, vous pouvez introduire celle-ci.

N.B. : ensuite, utiliser le prédéfini **exactly(Occ, Liste, Valeur)** qui permet d'imposer à *Liste* de contenir *Occ* fois la *Valeur*.

Par exemple, *exactly(2, L, 0)* permet d'avoir exactement 2 zéros (0) dans L.

$$\begin{array}{r}
 \begin{array}{rrr} X_1 & X_2 & X_3 \\ & * & \\ X_4 & X_5 & X_6 \end{array} \\
 \hline
 \begin{array}{rrr} X_7 & X_8 & X_9 \\ X_{10} & X_{11} & X_{12} \\ + & X_{13} & X_{14} & X_{15} \end{array} \\
 \hline
 X_{16} & X_{17} & X_{18} & X_{19} & X_{20}
 \end{array}$$

Solution :

Pour la liste des variables [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12, X13,X14,X15,X16,X17,X18,X19,X20],

On aura : [1, 7, 9, 2, 2, 4, 7, 1, 6, 3, 5, 8, 3, 5, 8, 4, 0, 0, 9, 6]

III.5.a Le code Prolog

```

mult(LD) :-
    fd_vector_min_max(0,9), % Optionnel. Permet de limiter la taille des domaines
    LD=[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,
        X11,X12,X13,X14,X15,X16,X17,X18,X19,X20],
    LD :: 0 .. 9,
    atmost(2,LD,0), atmost(2,LD,1), % ou atmost
    atmost(2,LD,2), atmost(2,LD,3),
    atmost(2,LD,4), atmost(2,LD,5),
    atmost(2,LD,6), atmost(2,LD,7),
    atmost(2,LD,8), atmost(2,LD,9), % ... suite ci-dessous

% Avec exactly(2,LD,0),... exactly(2,LD,9) la solution est plus lente à trouver
% atmost donne le même résultat car on a 20 variables et un domaine de taille 10.

Y#=100*X1+10*X2+X3,
Z1#=100*X7 +10*X8 +X9,
Z2#=100*X10+10*X11+X12,
Z3#=100*X13+10*X14+X15,
X6*Y #= Z1,
X5*Y #= Z2,

```

```
X4*Y #= Z3,  
100*X7 + 10*X8 + X9 +  
1000*X10+ 100*X11+ 10*X12 +  
10000*X13+ 1000*X14+ 100*X15 #=  
10000*X16+ 1000*X17+ 100*X18+ 10*X19+ X20,
```

```
labeling(LD).
```

Question :

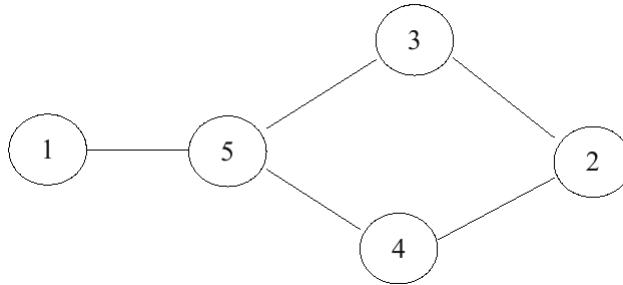
```
?- mult(L).
```

```
→ L = [1,7,9,2,2,4,7,1,6,3,5,8,3,5,8,4,0,0,9,6]
```

III.6 Coloration

Écrire un programme pour colorer le graphe suivant tel que deux noeuds adjacents n'aient pas la même couleur.

Remarque : Il y a des techniques pour minimiser le nombre de couleurs (voir poly, cours) mais pour cet exercice, un choix judicieux dans l'affectation des couleurs (dans l'énumération) peut aussi aboutir au même résultat.



Indications sur la spécification :

- 5 variables (X1..X5) à valeur dans 1..5.
- On se limite à n'utiliser que 5 couleurs au plus mais le prédicat principal peut être paramétrable par le nombre de couleurs.
- Pour trouver le nombre minimum de couleurs, on peut créer une équation sur les valeurs des variables (par exemple leur somme) et minimiser celle-ci.
Cette solution privilégie les couleurs de petit numéro.

Voir également *decoupe.pl* pour un exemple de *fd_minimize*.

III.6.a Le code

Le triplet (V, D, C) :

```

color([A, B, C, D, E]) :-
    L = [A, B, C, D, E],
    L :: 1 .. 5,
    A #\= E,
    E #\= C, E #\= D,
    B #\= C, B #\= D.
    
```

Questions :

color(L).

→ les variables conservent leurs domaines.

color(L), labeling(L). Solutions non optimales.

→ L = [1,1,2,2,3] , L = [1,1,2,2,4] , L = [1,1,2,2,5] , ...

color(L), labelingff(L).

→ L = [1,2,1,1,2] solution optimale puis des solutions non optimales.

L'énumération avec *labelingff* permet d'obtenir une solution optimum.

III.6.b Complément sur l'énumération (dans coloration)

Une énumération par la stratégie " first-fail " (*labelingff*) permet de donner, en premier, une valeur à la variable dont le domaine est de taille minimale. Ce qui minimisera naturellement le nombre de couleurs ici. L'écriture *labelingff(Liste)* est la même chose que le format général de *labeling* :

labeling([ff], Liste).

Si on souhaite que l'énumération traite d'abord les variables les plus contraintes (nombre de contraintes maximal), on utilise *labeling([constr], Liste_vars)*.

Une combinaison courante de *ff* et *constr* donne

labeling_ffc(Vars) ou *labelingffc(Vars)*

Autres options intéressantes : **énumérer tel que Expr soit maximisée**

labeling([maximize(Expr)], vars). Idem pour minimize.

ou *maxof(labeling(Vars), Expr).*

Exemple :

[A,B,C] :: 1..10, A #= B+C, B #> C, labeling([maximize(A)], [A,B,C]).
 $\rightarrow A=10, B=6, C=4.$

[A,B,C] :: 1..100, A #= B+C+D, B #> C, labeling([minimize(A), ff], [A,B,C]).
 $\rightarrow A = 1, B = 2, C = 1, D = -2$

A retenir : énumérations

labeling (Liste_vars)
labelingff(Liste_vars), labeling_ffc(vars),
labeling([constr], Liste),
labeling([minimize(Expr)], vars)

Voir la documentation Bprolog.

III.6.c Une autre stratégie indépendante d'une énumération particulière

Une autre stratégie pour minimiser le nombre de couleurs :

- trouver une solution (une liste de numéro de couleur dans 1..5),
- extraire le maximum des numéros de couleur distincts.
- minimiser ce maximum.

Cette dernière opération est directement possible dans les options de labeling.

```
colors_distinctes(L,N) :-  

    color(L), labeling([minimize(max(L))],L).  

    → L = [1,2,1,1,2]
```

III.6.d Encore une autre stratégie

Une autre stratégie pour minimiser le nombre de couleurs :

- trouver une solution (une liste de numéro de couleur dans 1..5),
- extraire l'ensemble (set) des numéros de couleur distincts.

- C'est la taille de cette liste (ensemble) que l'on minimise.

```
colors_distinctes(L,N) :-  
    color(L), labeling(L) , % obtenir une solution  
    setof(X,member(X,L),Set), % enlever les doublons (couleurs répétées)  
    length(Set,N). % calculer la taille de cet ensemble
```

Questions :

`colors_distinctes(Liste_couleurs, Taille).`

→ L = [1,1,2,2,3], Taille = 3

`Taille #> 0, minof(colors_distinctes(Liste_couleurs, Taille), Taille).`

→ L = [1,2,1,1,2], Taille = 2

Ici, toutes les solutions seront de taille 2.

III.6.e Une 3e stratégie

Une autre stratégie pour minimiser le nombre de couleurs :

- trouver une solution (une liste de numéro de couleur dans 1..5),
- faire la somme de cette liste.
- C'est cette somme que l'on minimise.

?- `color(L), labeling([minimize(sum(L))],L).`

→ L = [1,2,1,1,2]

Cette écriture est équivalente à :

?- `color(L), S#=sum(L), minof(labeling(L), S).`

Les inconvénients de cette solution :

La stratégie ci-dessus est du type Générer-Tester (tout comme la précédente) : les calculs de la somme (et de la taille et *setof* dans la méthode précédente) sont faits après avoir extrait une solution (après *labeling*).

Ce qui est peu efficace.

De plus, une somme minimale des couleurs ne garantie pas une solution optimale.

Par exemple, une solution utilisant les couleurs 1,2,3 donne une somme = 6 alors que la solution avec seulement 2 couleurs 3 et 4 sera considérée comme moins bonne.

De fait, minimiser la somme n'est acceptable que si les couleurs sont affectées dans l'ordre 1..N (pour écarter les solutions avec de grands numéros de couleur).

On peut alors avancer que la solution obtenue est optimale.

Preuve :

Soit un graphe à 3 noeuds et une solution avec trois couleurs 1, 2 et 3 et une autre solution avec deux couleurs 4 et 5.

La somme des trois couleurs 1+2+3 donnent une somme inférieure aux deux couleurs 4+3.

Cependant, les couleurs sont permutables : s'il y a une solution avec les couleurs 4 et 3, il y en a forcément une avec les couleurs 1 et 2.

III.7 Notes sur le calcul des extrema

On peut calculer ces valeurs extrêmes par la stratégie Branch & Bound :

- Trouver une première réponse pour **decoupe1/4**; notez la valeur de **Chutes** comme une référence
- Chercher d'autres solutions mais ne pas développer totalement toute autre solution si "sa valeur" de **Chutes** risque d'être moins 'bonne' (plus grande pour *minof*, plus petite pour un *maxof*).
- Si une telle solution existe, considérer la nouvelle valeur de **Chutes** comme référence et recommencer (sinon, ne pas changer la référence).
- Donner comme meilleure réponse celle de référence.

Ce qui peut être donné par le pseudo-algorithme :

my_Branch_and_Bound(Goal, Var):-

- 1- *Cout_ref* \leftarrow max couts % initialisation de *Cout_ref*
- 2- *Imposer la contrainte Var #< Cout_ref*
- Répéter*
- 3- *Appeler Goal*
- 4- *Si Goal réussit (avec Var #< Cout_ref)*
- 5- *Alors Noter la valeur de Var \rightarrow Cout_ref*
- 6- *Imposer Var #< Cout_ref*
- 7- *Sinon Var \leftarrow Cout_ref est le meilleur cout trouvé*

Jusqu'à faux;

"Noter la valeur de..." peut se faire par un passage de paramètre, utilisation des variables globales ou l'insertion d'un fait dans la base.

☞ Voir un exemple :

III.7.a Un exemple

Pour illustrer cette technique, on utilise le prédicat de coloration précédent adapté au calcul par la méthode Branch and bound.

Dans cette version, on trouve une solution dont le "cout" sera le nombre de couleurs distinctes utilisées.

`color([A, B, C, D, E]) :-`

```
L = [A, B, C, D, E],  
L :: 1 .. 5,  
A #\= E,  
E #\= C, E #\= D,  
B #\= C, B #\= D.
```

% Trouver une solution avec le n,ombre de couleurs utilisées

`colors_avec_nbr_couleurs(L,N) :-`

```
color(L), labeling(L), % obtenir une solution  
setof(X,member(X,L),Set), % enlever les doublons (couleurs répétées)  
length(Set,N). % calculer la taille de cet ensemble
```

% Branch and Bound Manuel.

`manual_bb(Solution_finale, Cout_Best) :-`

```
% trouver une solution qui donne un premier cout de référence (Cout_Ref)  
colors_avec_nbr_couleurs(Liste_temporaire, Cout_Ref)  
, suite_manual_bb(Liste_temporaire, Cout_Ref, Solution_finale, Cout_Best) % L'améliorer
```

`suite_manual_bb(_Solution_actuelle, Cout_actuel, Solution_finale, Cout_Best_final) :-`

```
Cout #< Cout_actuel % On impose que toute autre solution soit moins cher  
, colors_avec_nbr_couleurs(Liste, Cout) % Trouver une telle solution  
, ! % Si elle existe, essayer de l'améliorer  
, suite_manual_bb(Liste, Cout, Solution_finale, Cout_Best_final)
```

% On a fini par épuiser l'espace des solutions et on a un meilleur cout. On la récupère.
`suite_manual_bb(Solution_actuelle, Cout_actuel, Solution_actuelle, Cout_actuel) .`

Exemple :

?- `manual_bb(Solution_finale, Cout_Best).` → `[1,2,1,1,2], 2`

On trouve d'abord une solution utilisant 3 couleurs <`[1,1,2,2,3], 3`>

On demande ensuite à trouver une solution de cout < 3

La solution suivante trouvée (avec donc 2 couleurs) sera <`[1,2,1,1,2], 2`>

Il n'y a pas de solution avec 1 couleur (on impose un cout < 2)

On finit par conserver la solution <`[1,2,1,1,2], 2`>.

Ces détails sont visibles en mode trace.

IV Exercices à commencer en séance

- Trouver une solution pour un maximum des problèmes présentés ci-dessous. Ils sont de difficulté variable. Le chiffre au début de chaque sujet donne la note de la question, si elle est correctement traitée.
- Vous avez **un mois** pour rendre le **code** et un **rappor pdf** (expliquant le principe de votre méthode et le modèle mathématique utilisé).
- Travail individuel ou en binôme possible.
→ Pensez bien à inclure une trace d'exécution à la suite de chaque modèle proposé.
- Un conseil : faites-en le maximum avant le test de ce module !

IV.1 Problème de transport (2)

Ce problème consiste à déterminer l'approvisionnement en farine des magasins des quelques villes à partir d'un ensemble d'usines en minimisant les coûts de transport.

Les données du problème sont :

- Les points de départ des transports :
les usines à Marseille (U1), Lille(U2) et Bordeaux (U3)
- Les points d'arrivée :
les magasins sont à Paris (M1), Perpignan (M2), Nice (M3) et Lyon (M4)

Les coûts de transport :

Usine/magasin	M1	M2	M3	M4	Capa. Ui
U1	7	8	5	19	4
U2	3	7	2	11	6
U3	9	5	10	18	3
Demande Mj	2	2	6	3	13

Un couple (par exemple $(U1, M1) = 7$) signifie que le coût de transport d'une unité de U1 à M1 vaut 7.

Les demandes : Chaque magasin demande une certaine quantité (**demande**)

Les disponibilités : Chaque usine dispose d'un stock limité (la capacité Ui) et ne peut fournir une quantité supérieure à son stock.

Les contraintes :

- La demande de chaque magasin doit être satisfaite
- La disponibilité (stock) de chaque usine ne doit pas être dépassée

Problème :

Proposer un plan de transport où le coût total de transport doit être minimisé.

Note : Dans ce problème, la somme des demandes correspond à la somme des disponibilités. On peut donc exiger la satisfaction des demandes.

Si cela n'est pas le cas, alors :

- si $offre > demande$: créer une destination fictive de coût zéro pour la différence
- si $offre < demande$: créer une origine fictive de coût très grand pour la différence.

Indications : le principe d'une modélisation

On utilise :

U_{iMj} : la quantité de produit acheminée de l'usine U_i vers le magasin M_j .

Ainsi, pour satisfaire la demande de M_1 , il faut que la quantité fournie par U_1 (U_{1M1}) plus la quantité fournie par U_2 (U_{2M1}) plus la quantité fournie par U_3 soient ≥ 3 .

Idem pour les autres variables.

Également, U_1 peut fournir U_{1M1} à M_1 , U_{1M2} à M_2 , U_{1M3} à M_3 et U_{1M4} à M_4 .

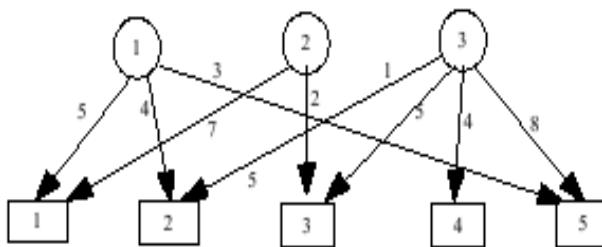
La somme de ces valeurs ≤ 4 (stock de U_1). Idem pour U_2 .

Enfin, les U_{iMj} représentent les quantités à transporter. Si U_1 fournit M_1 , chaque unité coûtera 5. On a donc les coûts des unités acheminées (à minimiser).

IV.2 Problème de localisation d'entrepôts (2)

Satisfaire les demandes des clients au moindre coût en implantant des entrepôts de marchandises dans diverses régions.

- Chaque entrepôt a un coût fixe d'ouverture et de maintenance :



On a les coûts 18, 10 et 28 pour les entrepôts E1, E2 et E3

- Un client n'est servi que par un seul entrepôt.

BUT : Servir tous les clients à moindre cout.

Trouver une configuration telle que le coût total soit inférieur ou égal à une limite (ici 60).

Une modélisation possible

On associe à chaque client ($i = 1..5$) à l'entrepôt J ($J=1..3$). Cela nous donne les variables C_{ij} , $j=1..3$, $i=1..5$.

$C_{ij} = 1$ si le client i est fourni par l'entrepôt j et 0 sinon.

N.B. : le domaine $\{0,1\}$ booléen permet de manipuler des valeurs comme des entiers (par exemple, dans une addition).

Contraintes :

- Un client n'est servi que par un seul entrepôt :

S'il existe $C_{ij}=1$, pour i et j donnés, cela veut dire que le client i est servi par l'entrepôt j . Dans ce cas, aucun autre entrepôt j' ne pourra servir ce même client.

.../..

Exemple : le client numéro deux ($i=2$) peut être servi par les entrepôts $j=1$ et $j=3$. On aura donc deux variables C_{21} et C_{23} . Si l'entrepôt 1 sert ce client (donc $C_{21}=1$), alors on doit avoir $C_{23}=0$. Cela peut être exploité par le prédéfini **only_one([C21,C23])**. Un autre manière de dire la même chose : **C21+C23 #= 1** (ou un XOR sur les deux ou **exactly**).

Propriétés de la solution (question de cohérence)

La solution doit être telle qu'un entrepôt non sélectionné ne doit pas figurer comme fournisseur d'un client. Cela nous conduit à utiliser des variables E_j , $j=1..3$.

La contrainte ci-dessus peut être traduit de la manière suivante :

Si E_j , $j=1..3 = 0$ pour un j donné alors il ne doit pas y avoir aucun client pour le même entrepôt :

Si $E_j=0 \rightarrow$ la somme (bool) des clients pouvant être servis par l'entrepôt j doit être = 0

Le Cout_total = le coût de transport pour chaque couple C_{ij} + le coût fixe d'ouverture et de maintenance de E_j .

$\text{Cout_total} \leq \text{Limite}$.

Si $C_{ij}=0$ ou $E_j=0$ alors le coût correspondant sera = 0

Rappel du schéma CSP (satisfaction de contraintes) :

- définition des variables
- définition des domaines
- définitions de contraintes (relations entre les variables)
- au besoin les contraintes (*a posteriori*) de vérification de la solution.

IV.3 Ordonnancement PERT (2)

On s'intéresse au problème d'ordonnancement suivant.

- il y a 10 tâches **A, B, C, D, E, F, G, H, I, J**
- à exécuter dans les durées respectives **5, 4, 3, 2, 1, 5, 4, 3, 2, 1**
- en respectant les contraintes de précédence

A avant B, C, D i.e. A doit être terminée avant le début de B (idem C et D)

B avant E

C avant F, G

D avant F

E avant H

F avant I

G avant I

H avant J

I avant J

a) Écrire un programme BProlog pour déterminer quelle est la date de démarrage au plus tôt de la tâche J ?

b) En fixant cette date de démarrage au plus tôt pour J, quelles sont les tâches qui ont une date flexible ?

IV.4 Ordonnancement disjonctif (2)

On ajoute au problème précédent la contrainte que les tâches E, F et G utilisent une même ressource, et ne peuvent donc pas s'exécuter en même temps.

- a) Écrire une requête BProlog pour déterminer quelle est la date de démarrage au plus tôt de la tâche J ?

IV.5 JobShop avec ressources et sans bénéfice (2)

Pour fabriquer 8 produits, on doit réaliser 8 tâches qui ont besoin d'une des machines M1 ou M2.

La durée de chaque tâche ainsi que la machine nécessaire sont données ci-dessous.

NB : il y a des contraintes disjonctives lorsque 2 tâches ne peuvent pas avoir lieu en même temps.

Exemple :

Tâche	Durée	Machine
1	2	1
2	3	1
3	1	2
4	2	2
5	3	1
6	4	2
7	2	1
8	3	2

Proposer un plan pour effectuer ces tâches .

IV.6 JobShop avec bénéfice (3)

En s'inspirant du problème précédent, proposer une solution pour le problème JobShop suivant :

NB : il y a des contraintes disjonctives lorsque 2 tâches ne peuvent pas avoir lieu en même temps.

Exemple :

La fabrication du produit Pi nécessite plusieurs phases (tâches) et plusieurs machines. La durée de chaque tâche est donnée.

Pour un produit, la séquence des tâches est ordonnée dans le temps.

Produit	Tâche	Durée	Machine	bénéfice
1	1	2	3	5
1	2	4	1	
1	3	3	2	
2	1	3	1	3
2	2	2	2	
3	1	1	3	2
3	2	3	1	

Problème :

1- Proposer un plan pour fabriquer une unité de ces 3 produits en minimisant la durée D de fabrication.

2- Pour une durée $D' \leq D$, quelles unités peut-on fabriquer telles que le bénéfice soit maximisé ?

N.B. : la question 1 impose une unité de chaque produit et donc Bénéfices= $5 + 3 + 2 = 10$.

Supposons que cette durée minimale $D = 10$. Pour la question 2 :

- Si $D' = D = 10$, on devrait avoir la même solution (Bénéfices=10)
- Si $D' = D - 1 = 9$, il faudra abandonner la fabrication d'un des produits.

→ Il faudra donc que pour la question 2, la fabrication de chaque produit devienne optionnelle et donc **ses contraintes exclues des calculs**.

IV.7 Problème d'emploi du temps (2)

Les organisateurs d'un congrès disposent de 3 salles et 2 jours pour organiser la tenue d'un congrès de 11 sessions (de A à K) d'une demi-journée.

Les (combinaisons de) sessions :

AJ, JI, IE, EC, CF, FG, DH, BD, KE, BIHG, AGE, BHK, ABCH, DFJ

ne peuvent pas se tenir en même temps car certains congressistes s'y sont inscrits.

→ C-à-d., les sessions A et J ne peuvent pas se tenir en même temps.

De plus certaines sessions doivent s'exécuter après d'autres :

J après E, K après D, K après F.

a) Écrire une requête Prolog qui détermine une date pour chaque session satisfaisant toutes les contraintes (en laissant libre l'affectation des salles).

b) dans un deuxième temps affecter les salles.

IV.8 Réunion (3)

Jean, Marie, Pierre et Jacques doivent se réunir. La secrétaire de Jean doit trouver un planning pendant la semaine de telle sorte que tout le monde puisse participer à cette réunion. La réunion dure 45 minutes et doit avoir lieu dans une salle souvent occupée.

Les temps libre de chacun est donné par le tableau suivant où <début , fin> donne le créneau horaire libre sur une échelle 0 .. 120 heures du lundi à vendredi :

lundi : 0 -> 24, **mardi : 25 -> 48**

mercredi : 49 -> 72

mardi : 25 -> 48

jeudi : 73 -> 96

vendredi : 97 -> 120

Les disponibilités :

Jean :

$\langle 9, 10 \rangle, \langle 13.5, 14.25 \rangle$	Lundi
$\langle 35.25, 35.75 \rangle, \langle 39, 41 \rangle$	mardi
$\langle 56, 58.25 \rangle$	mercredi
$\langle 84, 84.5 \rangle$	jeudi
$\langle 106, 107.25 \rangle$	vendredi

Marie :

$\langle 10, 10.5 \rangle, \langle 12.5, 13.25 \rangle,$	Lundi
$\langle 36.25, 37.75 \rangle, \langle 39, 41 \rangle,$	Mardi
$\langle 57, 58.25 \rangle,$	Mercredi
$\langle 80, 85.5 \rangle$	Jeudi

Pierre

$\langle 9, 11.5 \rangle, \langle 12.5, 15 \rangle,$	Lundi
$\langle 33, 35 \rangle, \langle 36, 42 \rangle,$	Mardi
$\langle 58, 59.25 \rangle,$	Mercredi
$\langle 80, 82 \rangle, \langle 92, 93.5 \rangle,$	Jeudi
$\langle 105, 109.25 \rangle$	Vendredi

Jacques

$<10.5, 15>$,	Lundi
$<32, 34>, <38, 40>$,	Mardi
$<56, 59>, <62, 63.75>$,	Mercredi
$<81.75, 83.25>$,	Jeudi
$<104, 106.5>$	Vendredi

La salle :

$<8, 12>$,	Lundi
$<38, 40>$,	Mardi
$<56, 57>$,	Mercredi
$<84, 86>$,	Jeudi
$<104, 106>$	Vendredi

But : Trouver une date pour cette réunion.

Indication : pour rester dans le domaine discret, travailler avec des minutes !

IV.9 Planification (5)

On construit une maison avec différentes tâches et un budget limité.

Il faut minimiser la total des durées pour emménager le plus vite Ordonnancement à faire avec contraintes de budget. Convention :

D_{xx} : Début tâche xx

F_{xx} : Fin tâche xx

L_{xx} : Durée tâche xx

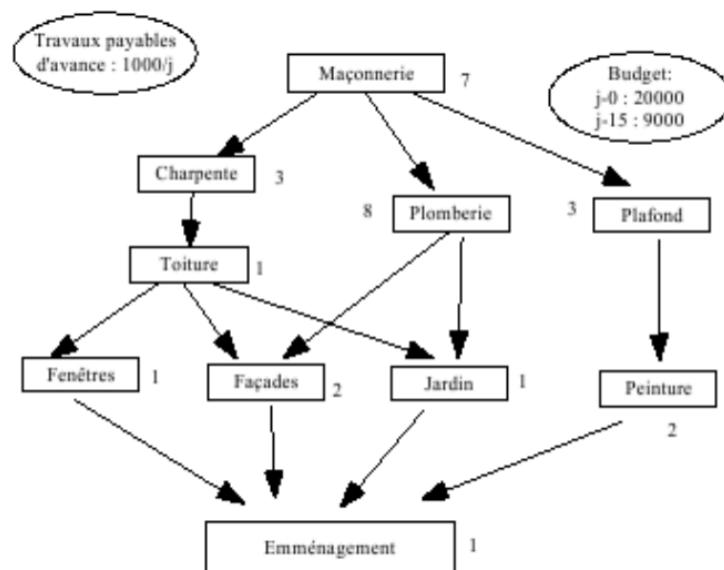
C_{xx} : Coût de xx = durée de xx * 1000 F à payer d'avance

Le coût total = somme des durées = 29 => 29000 F.

Les tâches : maçonnerie, charpente, toiture, plomberie, plafond, fenêtres, façades, jardin, pointure , emménagement

Il faut emménager le plus vite.

Indication : trouver une solution pour le cas sans les contraintes de budget puis la transformer en introduisant les contraintes de budget.



IV.10 Qui a Gagné la Médaille d'Or ? (3)

Jean, Pierre, Paul, André et Roland ont été sélectionnés pour les Jeux Olympiques. Ils sont alignés sur un podium pour recevoir les félicitations du président des Etats Unis d'Europe.

On sait que :

- Pierre a le bras cassé ;
- Une entorse a empêché l'un des grands favoris de terminer les jeux ;
- Paul fait de l'équitation ;
- Le spécialiste du saut à la perche boit du café ;
- Le buveur de jus de fruits a eu une insolation ;
- Le spécialiste du décathlon a attrapé un rhume ;
- Paul est à la droite de Roland ;
- Le buveur d'eau a eu une médaille d'or ;
- André boit du thé ;
- Celui du milieu fait de l'escrime ;
- Le premier à gauche a une cocarde à l'oeil ;
- Le buveur de lait est à côté de celui qui a fini quatrième ;
- Le buveur de thé est à côté de celui qui a eu une médaille de bronze ;
- Celui qui a une cocarde à l'oeil est à côté de Jean ;

→ Qui fait de la boxe ? Qui a eu une médaille d'or ?