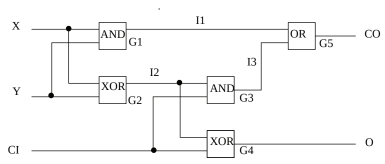
Modélisation par contraintes sur un domaine fini

- Contraintes booléennes
- Réification
- Contraintes complexes
- Choix dans la modélisation d'un problème
- Comparaison des modélisations différentes :
  - Efficacité
  - Flexibilité
- Exemple : ordonnancement

Contraintes booléennes



$$I1 \leftrightarrow X \& Y \wedge I2 \leftrightarrow X \oplus Y \wedge I3 \leftrightarrow I2 \& CI \wedge$$
$$O \leftrightarrow I2 \oplus CI \wedge CO \leftrightarrow I1 \vee I3$$

Ne pas confondre $\wedge$ avec & !

Arc-consistance pour contraintes booléennes

- Quand toutes les variables ont un domaine binaire : borne-consistance coïncide avec hyper-arc-consistance.
- Exemple : $z = x\&y$, $D(x) = D(y) = D(z) = [0, 1]$.
- Règles de propagation :
  - $x = 0 \Rightarrow z = 0$
  - $y = 0 \Rightarrow z = 0$
  - $x = 1, y = 1 \Rightarrow z = 1$
  - $z = 1 \Rightarrow x = 1, y = 1$
- Règles similaires pour $\vee$, $\leftrightarrow$, etc.

Contraintes booléennes en GNU Prolog

- GNU Prolog : Une *expression de domaine fini* est soit une variable de domaine fini, soit une contrainte de domaine fini.
- On peut construire des nouvelles contraintes en appliquant des prédicats booléennes à des expressions de domaines finis.
- On parle d'une contrainte *réifiée* : La véracité de la contrainte est utilisée comme une valeur booléenne.
- En général : La réification consiste à transformer ou à transposer une abstraction en un objet concret, à appréhender un concept, comme une chose concrète.

Contraintes réifiées

- Une contrainte réifiée `C #<=> B` attache une variable booléenne `B` à une contrainte simple `C`.
- Si `C` est vraie alors `B=1`.
- Si `C` est fausse alors `B=0`.
- Propagation *dans les deux sens*.

Exemple : Disjonction de contraintes

```
ou(X,X1,X2) :-
    fd_domain([B1,B2],0,1),
    (X#=X1 #<=> B1),
    (X#=X2 #<=> B2),
    B1+B2 #>= 1.

?- fd_domain(A,1,2), fd_domain(C,3,4),
   fd_domain(E,2,3),ou(A,C,E).
A = 2
C = _#25(3..4)
E = 2
```

Attention ce n'est pas

$$domain(A) = domain(C) \cup domain(E)$$

Exemple : ssi

```
ssi(X1,V1,X2,V2) :-
    fd_domain(B,0,1), X1 #= V1 #<=> B,  X2 #= V2 #<=> B.

| ?- ssi(0,0,Y,0).
Y = 0
yes

| ?- ssi(X,0,1,0).
X = _#39(1..268435455)
yes
```


Opérateurs booléens en GNU Prolog

#\ E	not E
E1 #<=> E2	E1 équivalent à E2
E1 #\<=> E2	E1 pas équivalent à E2
E1 ## E2	E1 ou-exclusif E2
E1 #==> E2	E1 implique E2
E1 #\==> E2	E1 n'implique pas E2
E1 #/\ E2	E1 et E2
E1 #\/\ E2	not (E1 et E2)
E1 #\/ E2	E1 ou E2
E1 #\\/ E2	not (E1 ou E2)

Exemple : Additionneur binaire

```
bitadd(X,Y,R,C) :-
    fd_domain([X,Y,R,C],0,1),
    R #<=> (X ## Y),
    C #<=> (X #/\ Y).
| ?- bitadd(1,0,R,C).
C = 0
R = 1
| ?- bitadd(X,1,0,C).
C = 1
X = 1
| ?- bitadd(X,Y,1,0).
X = _#0(0..1)
Y = _#18(0..1)
```


Solutionneur de contraintes booléennes

- Problème NP-complet
- Il existe des algorithmes efficaces pour certaines classes de formules qui fonctionnent souvent très bien en pratique (SAT solvers).
- Algorithmes probabilistes.
- En GNU Prolog : les contraintes booléennes sont un cas particulier des contraintes sur un domaine fini :
  domaine des variables booléennes = [0,1].

Contraintes complexes

- Permettent de modéliser plus succinctement
- Meilleure propagation, meilleure efficacité
- Exemple : alldifferent (en GNU Prolog : `fd_all_different`)
- Le choix des contraintes complexes disponibles, et leurs règles de propagation (donc la puissance du solutionneur de contraintes) font souvent la force des systèmes commerciaux de programmation par contraintes.

Contraintes complexes en GNU Prolog

- `fd_all_different(+L)` : contraint toutes les variables de la liste *L* à prendre des valeurs différentes. En GNU Prolog propagation seulement quand une variable devient instanciée; des règles de propagation plus fortes sont possibles (voir le cours sur les contraintes sur un domaine fini).
- `fd_element(I, List, X)` contraint `X` d'être égale au `I`-ème entier de la liste `List` (qui doit être une liste de valeurs entières).

Exemple element

```
?- fd_element(3,[1,2,4],E).
E = 4
| ?- fd_element(I,[1,2,4],2).
I = 2
| ?- fd_element(X,[0,1,2,3,4,5],Y), Y #> 3.
X = _#3(5..6)
Y = _#28(4..5)
| ?- fd_element(3,L,2).
uncaught exception:
   error(instantiation_error,fd_element/3)
```


La contrainte de cardinalité

- `fd_cardinality(+L, ?V)`, où L est une liste d'expressions de domaine fini, contraint X à être le nombre de contraintes en L qui sont satisfaites.
- `fd_cardinality([C1,...,Cn],X)` est équivalent à `B1 #<=> C1, ..., Bn #<=> Cn, X #= B1 + ... + Bn` où `B1, ..., Bn` sont de nouvelles variables.

Plus de contraintes de cardinalité

- `fd_at_least_one(L)` vraie quand au moins une contrainte de L est vraie. Équivalent à `fd_cardinality(L,X), X #>= 1`.
- `fd_at_most_one(L)` : au plus une contrainte de L vraie.
- `fd_only_one(L)` : exactement une contrainte de L vraie.

La contrainte cumulative

- Exemple d'une contrainte complexe et utile qui n'existe pas en GNU Prolog.
- `cumulative`$([S_1, \dots, S_n], [D_1, \dots, D_n], [R_1, \dots, R_n], L)$:
  - Problème d'ordonnancement : n tâches
  - S_i : début de la tâche i
  - D_i : durée de la tâche i
  - R_i : nombre de ressources nécessaires
  - L nombre de ressources disponibles

Exemple avec cumulative

- On a quatre personnes pour un déménagement
- On doit terminer en 50 minutes
- On a les objets suivants à déménager :

Objet	temps nécessaire	Personnes nécessaires
Piano	30	3
Chaise	10	1
Lit	15	3
Table	15	2

Solution avec cumulative

- Variable S_p : temps où on commence à bouger le piano.
- Similaire pour S_c, S_b, S_i.

$$\texttt{cumulative}([S_p, S_c, S_b, S_i], [30, 10, 15, 15], [3, 1, 3, 2], 4),$$
$$S_P + 30\# =< 50, S_c + 10\# =< 50,$$
$$S_b + 15\# =< 50, S_i + 15\# =< 50.$$

Problème d'affectation

- Quatre ouvriers w_1, w_2, w_3, w_4 et quatre produits p_1, p_2, p_3, p_4
- Affecter des ouvriers aux produits pour faire un profit ≥ 19
- Les profits sont donnés par

	p_1	p_2	p_3	p_4
w_1	7	1	3	4
w_2	8	2	5	1
w_3	4	3	7	2
w_4	3	1	6	3

1er modèle

- 16 variables booléennes B_{ij} qui signifient que ouvrier i est affecté au produit j
- $\bigwedge_{i=1}^{4} \sum_{j=1}^{4} B_{ij} = 1$
- $\bigwedge_{j=1}^{4} \sum_{i=1}^{4} B_{ij} = 1$
- $P = 7 * B_{11} + B_{12} + 3 * B_{13} + 4 * B_{14} + 8 * B_{21} + 2 * B_{22} + 5 * B_{23} + B_{24} + 4 * B_{31} + 3 * B_{32} + 7 * B_{33} + 2 * B_{34} + 3 * B_{41} + B_{42} + 6 * B_{43} + 3 * B_{44}$
- $P \geq 19$

2ème modèle

- Quatre variables correspondant aux ouvriers (quel domaine ?)
- Quatre variables correspondant aux profits par ouvrier (quel domaine ?)
- `alldifferent([W1,W2,W3,W4])`
- `element(W1,[7,1,3,4],WP1)`
`element(W2,[8,2,5,1],WP2)`
`element(W3,[4,3,7,2],WP3)`
`element(W4,[3,1,6,3],WP4)`
- $P = WP1 + WP2 + WP3 + WP4, P \geq 19$

3ème modèle

- Quatre variables correspondant aux produits (quel domaine ?)
- Quatre variables correspondant aux profits (quel domaine ?)
- `alldifferent([T1,T2,T3,T4])`
- `element(T1,[7,8,4,3],TP1)`
`element(T2,[1,2,3,1],TP2)`
`element(T3,[3,5,7,6],TP3)`
`element(T4,[4,1,2,3],TP4)`
- $P = TP1 + TP2 + TP3 + TP4, P \geq 19$

Quel est le meilleur modèle ?

- Le troisième est le plus efficace (pourquoi ?)
- Critères
  - Nombre de variables
  - Nombre de contraintes
  - Flexibilité
    - Comment ajouter la contrainte qu'on ne peut pas en même temps avoir ouvrier 1 affecté au produit 1 et ouvrier 4 affecté au produit 4 ?

Modéliser contraintes supplémentaires

Dans le premier modèle :

$$B_{11} + B_{44} \leq 1$$

Dans le deuxième modèle :

```
fd_domain([B11,B44],0,1),
element(W1,[1,0,0,0],B11),
element(W2,[0,0,0,1],B44),
B11 + B44 #<= 1
```

Modéliser contraintes supplémentaires

Ouvrier 3 doit travailler sur un produit de numéro plus grand que l'ouvrier 2.
Dans le deuxième modèle : $W_3 > W_2$
Dans le premier modèle :

$$
\begin{aligned}
& B_{31} = 0 \wedge B_{32} \leq B_{21} \wedge B_{33} \leq B_{21} + B_{22} \\
\wedge \quad & B_{34} \leq B_{21} + B_{22} + B_{23} \wedge B_{24} = 0
\end{aligned}
$$

Combiner les modèles

- Combiner les modèles en reliant les variables et leur valeurs dans chaque modèle
- p.e. $B_{13} = 1$ signifie $W_1 = 3$ signifie $T_3 = 1$
- Les modèles combinés peuvent être plus efficace grâce à la propagation d'information
- On suppose qu'on a une contrainte `ssi(V1,D1,V2,D2)` qui est vraie, si (`V1=D1` si et seulement si `V2=D2`)

Modèle combiné (Modèles 2 et 3)

```
alldifferent([W1,W2,W3,W4])   alldifferent([T1,T2,T3,T4])
element(W1,[7,1,3,4],WP1)     element(T1,[7,8,4,3],TP1)
element(W2,[8,2,5,1],WP2)     element(T2,[1,2,3,1],TP2)
element(W3,[4,3,7,2],WP3)     element(T3,[3,5,7,6],TP3)
element(W4,[3,1,6,3],WP4)     element(T4,[4,1,2,3],TP4)
P #= WP1 + WP2 + WP3 + WP4    P #= TP1 + TP2 + TP3 + TP4
P #>= 19
ssi(W1,1,T1,1) ssi(W1,2,T2,1) ssi(W1,3,T3,1) ssi(W1,4,T4,1)
ssi(W2,1,T1,2) ssi(W2,2,T2,2) ssi(W2,3,T3,2) ssi(W2,4,T4,2)
ssi(W3,1,T1,3) ssi(W3,2,T2,3) ssi(W3,3,T3,3) ssi(W3,4,T4,3)
ssi(W4,1,T1,4) ssi(W4,2,T2,4) ssi(W4,3,T3,4) ssi(W4,4,T4,4)
```


Exemple: Ordonnancement

- Un ensemble de tâches est donné
  - avec des préséances (des tâches doivent être terminées avant des autres)
  - et des ressources partagées (des tâches ont besoin de la même machine)
- Déterminer un bon ordonnancement, donc
  - les contraintes sont satisfaites
  - le temps global est minimisé
- Ici nous fixons uniquement une limite.

Exemple

On représente les données comme une liste de tâches

```
task(nom,duree,[noms],machine)

problem([task(j1,3,[],m1),
 task(j2,8,[],m1),
 task(j3,8,[j4,j5],m1),
 task(j4,6,[],m2),
 task(j5,3,[j1],m2),
 task(j6,4,[j1],m2)]).
```


Programme

- Forme du programme
  - Définir les variables: `makejoblist`
    - variables: Temps de début de chaque tâche
    - liste: `job(nom,duree,StartVar)`
  - Contraintes de préséances : `precedences`
  - Contraintes de machines : `machines`
  - Labeling : `labeltasks`
    - prendre des variables de la liste des jobs et donner une valeur

Programme

```
schedule(Data, End, Joblist)   :-
         makejoblist(Data, Joblist, End),
         precedences(Data, Joblist),
         machines(Data, Joblist),
         labeltasks(Joblist).

makejoblist([],[],_).
makejoblist([task(N,D,_,_)|Ts], [job(N,D,TS)|Js], End)  :-
        fd_domain(TS,0,End),
        TS + D #=< End,
        makejoblist(Ts, Js, End).

getjob(JL, N, D, TS)  :- once(member(job(N,D,TS), JL)).
```


Programme: préséances

```
precedences([],_).
precedences([task(N,_,Pre,_)|Ts], Joblist)  :-
        getjob(Joblist, N, _, PostStart),
        prectask(Pre, PostStart, Joblist),
        precedences(Ts, Joblist).

prectask([],  _, _).
prectask([Name|Names], PostStart, Joblist)  :-
        getjob(Joblist, Name, D, Start),
        Start + D #=< PostStart,
        prectask(Names, PostStart, Joblist).
```

Programme: machines

```
machines([], _).
machines([task(N,_,_,M)|Ts], Joblist)  :-
        getjob(Joblist, N, D, Start),
        machtask(Ts, M, D, Start, Joblist),
        machines(Ts, Joblist).
```


Programme: machines (2)

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

machtask([task(SN,_,_,MO)|Ts], M, D, TS, Joblist)  :-
             (M = MO ->
                 getjob(Joblist, SN, SD, STS),
                 exclude(TS, D, STS, SD)
             ; true ),
            machtask(Ts, M, D, TS, Joblist).

exclude(AStart,AD,BStart,BD)  :-  BStart + BD #=< AStart.
exclude(AStart,AD,BStart,BD)  :-  AStart + AD #=< BStart.
```


Labeling

```
labeltasks([]).
labeltasks([job(_,_,TS)|Js])   :-
     fd_labeling(TS),
     labeltasks(Js).
```


Exécuter ordonnancement (1)

```
?- problem(Problem), End = 20, schedule(Problem,End,LJobs).
```

`makejoblist` : construire les contraintes initiales et ajouter des contraintes

```
[job(j1,3,TS1),job(j2,8,TS2),job(j3,8,TS3),
 job(j4,6,TS4),job(j5,3,TS5),job(j6,4,TS6)]
```

Exécuter ordonnancement (2)

Domaines initiales des variables:

`D(TS1)=[0..17], D(TS2)=[0..12], D(TS3)=[0..12]`
`D(TS4)=[0..14], D(TS5)=[0..17], D(TS6)=[0..16]`

Raison : starttime $+$ durée $\leq$ temps limite pour toute tâche

Exécuter ordonnancement (3)

`precedences` : ajoute des contraintes et change les domaines

```
TS1+3 #=< TS5   TS1+3 #=< TS6
TS4+6 #=< TS3   TS5+3 #=< TS3
```

`D(TS1)=[0..6], D(TS2)=[0..12], D(TS3)=[6..12]`
`D(TS4)=[0..6], D(TS5)=[3..9], D(TS6)=[3..16]`

Exécuter ordonnancement (4)

`machines` : ajoute des choix de contraintes, change les domaines

```
TS2+8 #=< TS1 ou TS1+3 #=< TS2
TS3+8 #=< TS1 ou TS1+3 #=< TS3 ...
```

`D(TS1)=[0..0], D(TS2)=[3..4], D(TS3)=[12..12]`
`D(TS4)=[6..6], D(TS5)=[3..3], D(TS6)=[12..16]`

Labeling

Dans ce cas on peut choisir la première valeur pour chaque variable

`D(TS1)=[0..0], D(TS2)=[3..3], D(TS3)=[12..12]`
`D(TS4)=[6..6], D(TS5)=[3..3], D(TS6)=[12..12]`

Solution trouvée