

BE : Recherche Opérationnelle & Programmation avec Contraintes

Optimisation Combinatoire & Programmation Mathématique

École Centrale de Lyon
Département Mathématiques-Informatique

UMR LIRIS - CNRS
Alexandre.Saidi@ec-lyon.fr

Octobre 2024

Installer Minizinc

- On installera le logiciel (MiniZinc 2) et un éditeur intégré (MiniZincIDE).
- Informations générales sur le site <http://www.minizinc.org/>

Pour toutes les plateformes (Windows, Mac, Linux) :

- Sur le site <https://www.minizinc.org/software.html>, télécharger votre version.
- Décompresser l'archive téléchargée en cliquant sur l'archive :

Pour **Linux** : décompresser dans une fenêtre "Terminal" par la commande :
`tar xzf "nom-de-archive.tgz"`

Puis placez vous dans ce répertoire et lancer "MiniZincIDE".

Pour **MacOsX** : cliquer sur le fichier télécharger et suivre ce qui s'affiche !

Pour **Windows** : cliquer sur le fichier télécharger et suivre ce qui s'affiche !

→ Si problème, demandez à l'enseignant !

Installer Minizinc (suite)

- Vous aurez besoin du tutorial de Minizinc.
- ☞ Le fichier *minizinc-tute.pdf* : tutoriel pour apprendre à utiliser *minizinc*.
Fichier à télécharger à ("xxx" est la version de Minizinc)
[https://www.minizinc.org/doc-xxx/en/MiniZinc Handbook.pdf](https://www.minizinc.org/doc-xxx/en/MiniZinc%20Handbook.pdf)
- ☞ Une page intéressante : <http://www.hakank.org/minizinc/>
- ☞ **Test pour toutes les plateformes :**
 - Charger (par "Open" dans le menu "File") le fichier d'exemple *demo_color.mzn*
puis cliquer sur le triangle (Run) pour exécuter *minizinc* sur cet exemple.
 - Si tout va bien, vous devriez avoir la solution.

Installer Minizinc (suite)

- ☞ Pour changer de solveur, aller dans l'onglet "configuration" et préciser des paramètres : par exemple "COIN-BC" pour une résolution dans \mathbb{R} (avec float / int)
 - Relancer la résolution du même fichier et obtenir les résultats.
- ☞ Pour ajouter d'autres solveurs à Minizinc (ou utiliser Minizinc dans d'autres applications) : <https://www.gecode.org/download.html>
 - Vous pourrez récupérer différents solveurs pour votre plateforme.
- Dans certains cas, on doit recompiler depuis les sources récupérées sur la même page.

Éléments de syntaxe

Variables "paramètres" : reçoivent une valeur (le mot **var** absent)

```
int: i=3;           % un paramètre (car initialisé)
par int: i=3;       % un paramètre (le mot "par")
int: i; i=3;        % déclaration + initialisation
```

- ➔ Commentaire : utiliser % ou / * .. * /
- ➔ Si on oublie la valeur constante, celle-ci sera demandée à l'exécution.
- ➔ Si *minizinc* exécuté au clavier, on peut préciser (p. ex.) -D "i=3"

Variables de décision : présence du mot clef **var**

```
var 0..4: i;           %déjà contrainte par un domaine
var {0,1,2,3,4} : j; % idem
var int: k;

% On exprime des contraintes sur ces variables
constraint k >= 0 /\ k <= 4 ;
constraint j > k /\ k > i;
% Solution : i = 0; j = 2; k = 1;
```

Éléments de syntaxe (suite)

☞ La déclaration : `var {0,1,2,3,4}: i;`
 permet à la variable i de prendre une des 5 valeurs parmi $\{0,1,2,3,4\}$.

Une déclaration équivalente sera : `var 0..4 : i;`

A noter : la première forme permet d'écarter certaines valeurs que la 2e ne permet pas.

Par exemple : `var {0,1,4}: i;`

☞ La déclaration suivante définit l'intervalle $i=0..4$:

```
var set of int : i = {0,1,2,3,4} ;
```

→ "i" n'est plus une variable mais un intervalle (où le mot clef "*var*" est inutile puisque la valeur de i est fixée).

→ On peut par exemple utiliser "i" comme indice d'un tableau.

Éléments de syntaxe (suite)

Un Exemple d'utilisation :

```

var int: var_i; % i est une var de décision

% Ci-dessous, le mot 'var' est inutile et const_intervalle_j
  représente l'intervalle 0..4
var set of int : const_intervalle_j = {0,1,2,3,4} ;

var {3, 5, 7, 11}: var_k_contrainte_by_a_set;
var 0..4: var_m_contrainte_by_intervalle;

constraint var_i >= 0;
constraint var_i <= 4;

solve satisfy;

% TRACE :
% on aura (les variables prennent leur 1ere valeur)
% var_i = 0;
% var_k_contrainte_by_a_set = 3;
% var_m_contrainte_by_intervalle = 0;
-----
Finished in 35msec

```

Types

Types de base :

- **Entiers** : *int*
ou *range 1..n*
ou utilisé dans *set of int*

```
int red = 1;  
set of int: Colors = 1..3;
```

- **Réels** : *float* ou *range 1.0 ...10.0* ou *set of float*
- **Booléen** : *bool*

Type ensemble

- **Ensemble** : *set of type*

```
set of int: Colors = 1..3;
var set of 1..10 : zz;
```

- ➔ Passage d'un *Set* vers un *Array* (tableau) : `set2array`
- Opérateurs sur les ensembles (comme dans $\{1,2\} \cup \{3,4\}$)
in, union, intersect, subset, superset, diff, symdiff, card
- Un (autre) exemple d'*ensemble* : x est "nbr premier" et $2x \geq 10$

```
set of int : prime = {1,2,3,5,7,11}; % "{..}" car "set of ..."
var int : x;

constraint
  x in prime /\ 2*x >= 10;    % "/" = conjonction = AND

solve satisfy;

% x = 5
% puis 7 ... (si plusieurs solutions demandées)
```

Type énuméré

Exemple :

```

set of int: POS = 1..6;
array[POS] of var PERSON: order;
enum PERSON = {anne, bob, cal, dan, edna, fred};

constraint order[fred] > order[edna];

solve satisfy;

% order = array1d(1..6 , [anne, anne, anne, anne, anne, bob]);

```

... ~→

Types tableau (et string)

Types non basiques (structurés) :

- **String** : `string` (ne peut pas être une variable de décision)
- **Tableau / tenseurs** : `array [range1, range2, ..., range6] of type`

• Exemple 1 :

```
array [0..2] of var 1..5 : v;           % Accès aux tableaux par V[
1]
array [1..5, 1..5] of var 0..2 : M ;    % ou par M[2,3]
```

• Exemple 2 : *Array* avec l'utilisation d'un "intervalle" pour initialiser un tableau

```
set of int : intervalle = 1..11;
array[20..30] of int: i = array1d(20..30, intervalle);

array[int, int] of float: z =
  array2d(1..10, 1..10, [ 0.0 | i, j in 1..10 ]);
```

☞ Voir plus loin "array en compréhension".

Types tableau (et string) (suite)

• Exemple 3 :

```

set of int: Colors = 1..3;
array[Colors] of string: name = ["red", "yellow", "blue"];

array[1..10] of var int : tab;
solve satisfy;

output [ show(name), show(tab) ];
% ["red", "yellow", "blue"]
% [-2147483646, -2147483646, -2147483646, -2147483646, -
  2147483646, -2147483646,
%   -2147483646, -2147483646, -2147483646, -2147483646]

```

Les sorties

- *output [liste de strings]* (chaînes de caractères)
 - ➔ **show(variable)** : écrire une valeur
 - ➔ concaténation avec "++" :
 "bella" ++ " stella" : donnera "bella stella"

Exemple :

```
...
output [ "Voici les variables \n", "i= ", show(i) ]
++ [ "\n j= ", show(j) ]
++ [ "\n k= ", show(k), "\n m= ", show(m) ] ;
```

• Exemple 2 : Strings et "output" :

```
% Concaténer avec ++, fusionner les arrays avec le string
% Convertir les expressions en une chaîne avec "show"
array [ 1..3 ] of var int : x ;
string : s = "Résultat = " ++ join(" - ", [show(x[i]) | i in 1..3
]);
% x = [ -2147483646, -2147483646, -2147483646 ] ;
```

Opérateurs

- Pour les réels : * / + -
- Pour les entiers : * **div** **mod** + -
- Comparaisons : == != < > >= <=
- *Autres* :
 abs, acos, asin, atan, ceil, concat, cos, cosh, exp, floor,
 ln, log, log2, log10, pow, round, sin, sinh, sqrt , tan , tanh, ...
- Opérateurs logiques :
 \rightarrow (*implication*), \leftarrow (*only if*), \leftrightarrow (*if and only if*),
 \vee , \wedge , *not*, *forall*, *xorall*, *exists*

Conversions

Éléments de conversion :

- Pas de conversions automatique entre les entiers et les réels :
`int2float(entier)` pour convertir.
 - De même : `bool2int`, `ceil` (excès), `round` (excès), `floor` (défaut) ...
 - En version 2 : la conversion de `bool` en `int` se fera automatiquement.
 - L'expression `2.5+2` est acceptée (le réel 4.5) mais cela dépendra du solveur (MIP?).
 - Passage d'un *Set* vers un *Array* (tableau) : `set2array`
- ☞ Dans les versions récentes, certaines conversions sont devenues automatiques.

Itérations

- Itération via des fonctions : **forall**, **exists**
- ☞ Les fonctions **sum**, **product**, **min**, **max** enferment une itération implicite
- Exemple **sum** :

```
sum(i, j in 1..5) (i*j)
```

équivalent à

```
sum([i*j | i, j in 1..5])
```


Itérations (suite)

- *forall* permet d'imposer une contrainte à **tous** les éléments (par exemple) d'un tableau

- Exemple (**forall**) : `forall (i,j in 1..10 where i<j) (a[i] != a[j]);`

qui est équivalent à (à l'aide d'un "tableau en compréhension") :

```
forall ([a[i]!=a[j] | i,j in 1..10 where i<j]);
```

- Exemple 2 (**forall**) :

```
forall (i in 1..9) (x[i+1] = x[i] + y[i]);
```

Qui est équivalent à :

```
forall ([x[i+1] = x[i] + y[i] | i in 1..9]);
```

- Ici, `forall(array[int] of var bool : B)`
est une fonction qui renvoie la conjonction de toutes les expressions dans B.
- Le résultat de "forall" dépend donc du contenu de B.

Itérations (suite)

- **exists** permet d'imposer une contrainte à **au moins un** des éléments (par exemple) d'un tableau.

Par exemple : `exists(i in 1..10)(t[i] > 10);`

→ On vérifie qu'au moins un des éléments du tableau `t` est `> 10`.

☞ Comparez avec `forall(i in 1..10)(t[i] > 10);`

- Exemple (**exists**) : (Mettre le solveur sur Chuffed ou G12fd)

```
int: x = 3;
int: y = 4;
var int : a;
predicate smallVal(var int:y) = -x <= y /\ y <= x;
predicate p(int: u) = exists(x in 1..u)(smallVal(x) -> a = x);
    % "->" : implication logique
constraint p(x);

% a=1
```

Structure d'un programme

• Exemple :

```

include "globals.mzn";    % si besoin (des contraintes globales)

% un paramètre entier (sa valeur doit être fixée)
int: n;
var 1..n: x;               % un entier (variable de décision)
array[1..n] of var 1..n: y; % tableau de var de décision

constraint sum(y) <= x;    % contrainte

solve satisfy;             % trouver une solution

% écrire une liste de strings
% \ (y) transforme la valeur de y en un string
output ["Solution:\n", "x = ", show(x), " et y = \ (y)"];

/* au lancement, la valeur de n : 10
   x = 10 et y = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
*/

```

→ N.B. : les deux formes de commentaire.

Type-inst

- Rappel de quelques types de base (ici, des constantes)

```
int : i;  
3..5 : j;           % entier  
  
float : f;  
3.0..5.0 : g;      % réel  
  
bool : b;          % bool  
  
set of int : s;    % set  
set of 3..5 : t;
```

Type-inst (suite)

- *Type-insts* : variables de décision (elles seront **globales**) :

```

var int: x;           % déclare la variable x
                      % (aussi avec : float, bool, set of int)

var 3..7: y;          % declare variable with domain (idem avec
  3.0..7.0)

var set of 10..20: s; % déclare un ens. de variables (Il faut "
  set of int" fixe !)

array[1..4,1..10] of var 0.0..100.0: f; % déclare des
  variables d'un 2d-array of float

```

- *Type-insts* : variables optionnelles

```

var opt 1..10: ox; % variable entier optionnelle ox (peut
  valoir 1..10 OU être absente)

```

Contraintes

- Contraintes de base :

```

constraint x = y;      %ou "=". On utilise "=" plutôt hors des contraintes
constraint x != y;     % différence
constraint x < y;
constraint x <= y;
constraint x > y;
constraint x >= y;

```

- Contraintes logiques avec connecteurs (opérateurs) logiques :

```

% conditionnels
int: d = if i > 10 then a elseif i > 0 then b else c endif;

constraint
  if x < y then y < z else y > z endif; % "else" et "endif" obligatoires

constraint
  if x < y then y < z else true endif; % Si on n'a pas de "else", mettre 'true'

constraint x < y  $\vee$  y != z;      % "or"
constraint x < y  $\wedge$  y != z;     % "and"
constraint x < y  $\rightarrow$  y != z;     % implication
constraint x < y  $\leftrightarrow$  y != z; % équivalence (if-and-only-if)
constraint x < y  $\leftarrow$  y != z;  % only-if
constraint not (x < y  $\wedge$  y > z); % négation

```

👉 **bool2int** transforme un Booléen en 1 (true) ou 0 (false)

Contraintes (suite)

- Contraintes sur les ensembles

```
constraint s subset t;           % opération subset non-strict (avec =)
constraint s intersect t subset w; % intersection
constraint s union t subset w;   % union
```

- On peut utiliser des contraintes globales prédéfinies (ou définies par nous)

```
constraint all_different(x);      % appel du prédicat globale "all_different"
constraint mydiv(x,y) = 2;        % appel de la fonction "mydiv"
```

- 👉 On peut définir les nôtres!

```
predicate no_overlap(var float : offset, var int: x, var int: y) =
    offset + x < y \ / offset + x > y;

function var float: average(array[int] of var int: x) = sum(x) / length(x);

array[1..10] of var 1..50 : tab;

constraint
    forall(i,j in 1..10 where i < j) (no_overlap(average(tab), tab[i], tab[j]));

solve satisfy;

output [show(tab)];           % Donne un tableau de 1
```

Array / Set en compréhension

- Array (et Set) en Compréhensions (= générateurs) :

```
% La taille du tableau est "calculable"
% Création de array [1,2,3,4,5,6,7,8,9,10]
array[int] of int: a = [ i | i in 1..10 ];

% Création de array [3, 6, 9]
array[int] of int: a = [ i | i in 1..10 where i mod 3=0 ];
```

Deux contraintes utilisant des **array of comprehension**.

```
constraint forall (i,j in 1..n where i<j) (x[i] != x[j]);
% équivalent à
constraint forall ([ x[i] != x[j] | i,j in 1..n where i<j ]);
```

Array à 2 dimensions (matrice) en compréhension

```
...
array[int,int] of float: z =
  array2d(1..10, 1..10, [ 0.0 | i,j in 1..10 ]);
```


Array / Set en compréhension (suite)

D'autres exemples de *set/array* en compréhension :

- Exemple de **Set en compréhension** :

$$\{i+j \mid i, j \text{ in } 1..3 \text{ where } i < j\} = \{1+2, 1+3, 2+3\} = \{3, 4, 5\}$$

- Exemple de **Array en compréhension** :

```
set of int: cols = 1..5;
set of int: rows = 1..2;
array [rows, cols] of int: c= [|
                                250, 2, 75, 100, 0, |
                                200, 0, 150, 150, 75 |];

% Array en écompréhension :
b = array2d(cols, rows, [c[j, i] | i in cols, j in rows]);
```

Stratégies de résolution

Stratégie par défaut :

- Un exemple simple (utilisant les ensembles / tableaux) :

```

var set of 1..2: x;
array[1..2] of var bool: y;

constraint
  x = {1} /\
  forall(i in x)(y[i]);

solve satisfy;

output [ "x = ", show(x), "\n", "y = ", show(y), "\n", ];

% 2 solutions demandées :
x = 1..1
y = [true, false]
-----
x = 1..1
y = [true, true]

```

Stratégies de résolution (suite)

- Exemple 2 :

```

array[1..3] of set of 1..3: a = [{1,2}, {3}, {1,3}];
var 1..3: i;

constraint
    card(a[i]) > 1;

solve satisfy;

% solution
i = 1; % a[1] satisfait la contrainte

% Et si on demande toutes les solutions :
i = 1;
i = 3; % a[3] satisfait la contrainte

```

☞ Voir plus loin pour la minimization / maximization d'une solution.

Stratégies de résolution (suite)

- Stratégies de recherche de solutions plus élaborée :

```
% optimisation
solve minimize sum(x);

solve :: int_search([x,y,z], input_order, indomain_min, complete)
    satisfy;
    % Stratégie de recherche de solution ..
    % ... via la sélection de variables :
    % input_order, first_fail, max_regret, smallest
    % ... ou via la sélection des valeurs (du domaine)
    % indomain_min, indomain_max, indomain,
    % indomain_split, indomain_reverse_split
    % De même pour bool_search, set_search, float_search
```

- Ou en ordonnant les recherches sur les variables :

```
solve :: seq_search([int_search(x, first_fail, indomain,
                                complete),
                    int_search(y, input_order, indomain_min, complete)])
    satisfy;
% D'abord, chercher x, Puis y
```

Notre premier programme : affichage !

Exemple : simple affichage !

- Créer le fichier *hello.mzn* avec le contenu suivant :
- ☞ Si vous n'utilisez pas l'éditeur de Minizinc, évitez les outils de PAO et le texte formaté comme *Word* et *OpenOffice*

```
% Pas un pb. d'optimisation, mais c'est un exemple pour Afficher seulement Hello
solve satisfy;
output ["Hello !", "\n"];
```

Sauvegarder puis "RUN" sous MinizincIDE.

Ou dans une fenêtre "Terminal", lancer `minizinc hello.mzn`

- Vous devriez voir le message "Hello !" s'afficher.
- ☞ Sans "output", Minizinc affiche les valeurs des variables de décision.
- "output" permet une mise en forme des résultats.

Exemple triangle : contraintes de base

- Soit le problème simple (**triangle.mzn**) que vous pouvez créer à l'aide d'un éditeur de texte (ou sous *mzIDE* ou *MiniZincIDE* ou tout autre GUI) :

☞ Notez **solve maximize c;**

```
var int : x1 ;
var int : x2 ;
var int : c ;

constraint
  x1 >= 0 ∧ x2 >= 0
  ∧
  c = x1 + x2 ∧ c <= 1;

solve maximize c;

output ["x1 = ", show(x1), " x2 = ", show(x2), " c = ", show(c), "\n"]; % affichage bruts.

% Résultat : x1 = 1 x2=0 c=1
```

- En ligne de commande : `minizinc triangle.mzn` ou `mzn-g12fd triangle.mzn`

Exemple triangle : contraintes de base (suite)

On peut minimiser / maximiser une expression, pas seulement une variable.

➔ **Maximiser** $x_1 + x_2$ sans borne supérieure précisée (pas une bonne idée!):

```
var int : x1 ;
var int : x2 ;

constraint
  x1 >= 0
  ∧
  x2 >= 0
;

solve maximize x1+x2;

output ["x1 = ", show(x1), ", x2=", show(x2), "\n"]; % affichage des résultats bruts.

% résultat après un certain temps : x1 = 10 000 000 , x2=0
```

☞ Valeur maximale par défaut d'un entier : env. 10^7

Satisfy ou maximize

- Dans les deux versions ci-dessous, on voit la différence entre *satisfy* (résolution sans optimisation) et *maximize* (optimisation).

```
var 0..5 : x;
constraint x*x = x+x;

solve satisfy;
output ["x= ", show(x)]

% On obtient x = 0
```

```
var 0..5 : x;
constraint x*x = x+x;

solve maximize x;
output ["x= ", show(x)]

% On obtient x = 2
```


Coloration : contrainte !=

Exemple tiré du tutoriel de Minizinc.

- On souhaite affecter une couleur à chaque région de la carte ci-dessous en respectant la contrainte suivante :

→ Deux régions limitrophes n'auront pas la même couleur.

👉 Plus tard, on optimisera le nombre de couleurs !

```
% Colouring Australia using nc colours
int: nc = 3;
var 1..nc: wa;
var 1..nc: nt;
var 1..nc: sa;
var 1..nc: nsw;
var 1..nc: v;
var 1..nc: t;
var 1..nc: q;
```



Coloration : contrainte \neq (suite)

```

constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;

solve satisfy;

output [ "wa=", show(wa), "\t nt=", show(nt),
        "\t sa=", show(sa), "\t n", "q=", show(q),
        "\t nsw=", show(nsw), "\t v=", show(v), "\t n",
        "t=", show(t), "\t n" ];

/* Résultats : minizinc coloration.mzn
wa=1 nt=3 sa=2
q=1 nsw=3 v=1
t=1
-----
Avec un autre solveur
wa=2 nt=3 sa=1
q=2 nsw=3 v=2
t=1
*/

```

Coloration : contrainte \neq (suite)

Le même exemple avec *alldifferent* :

```
include "alldifferent.mzn";

int: nc = 3;
var 1..nc: wa;
var 1..nc: nt;
var 1..nc: sa;
var 1..nc: nsw;
var 1..nc: v;
var 1..nc: t;
var 1..nc: q;
constraint alldifferent([wa,nt,sa]);
constraint alldifferent([q,nt,sa]);
constraint alldifferent([v,nsw,sa]);
constraint alldifferent([q,nsw,sa]);

solve satisfy;

output ["wa=", show(wa), "\t nt=", show(nt), "\t sa=",
        show(sa), "\n", "q=", show(q), "\t nsw=",
        show(nsw), "\t v=", show(v), "\n", "t=", show(t), "\n"];

% wa=1 nt=3 sa=2 q=1 nsw=3 v=1 t=1
```



Déclaration et initialisation de vecteur/matrices


- Dans l'exemple suivant, on déclare une matrice puis un vecteur reçoit la somme de chaque ligne de la matrice puis on affiche le min et le max de ces sommes.

```
% Nous aurons une matrice 10 x 2 d'entiers
par int : n;          % "par" : on prévient qu'on a un "paramètre" (défini ci-dessous)
par array[1..n, 1..2] of int : points;

% On utilise un vecteur pour stocker la somme de chaque ligne de la matrice "points"
array [1..n] of var int : sommes;
constraint forall(i in 1..n) (sommes[i]=points[i,1]+points[i,2]);

solve satisfy;      % pas de fonction objective

% On veut le min et le max du vecteur "sommes"
output["min= ", show(min(sommes)), ", ", max= ", show(max(sommes)), "\n"];
%-----
% DATA
n = 10;
points = array2d(1..n, 1..2, [
    10, 6,
    5, 7,
    2, 9,
    5, 4,
    3, 5,
    5, 7,
    4, 1,
    2, 2,
    0, 7,
    8, 6
]);
% Trace : min= 4, max= 16
```

 Ici, l'initialisation est faite via la fonction `array2d`.

Déclaration et initialisation de vecteur/matrices (suite)

- ON pourrait initialiser la matrice "points" initialisée d'une 2e façon :
 → Remarquer l'ajout des '|' (pour délimiter les lignes de la matrice).

```
% Nous aurons une matrice 10 x 2 d'entiers
```

```
int : n=10;
```

```
array[1..n, 1..2] of int : points= [|
```

```
  10, 6 |
```

```
  5, 7 |
```

```
  2, 9 |
```

```
  5, 4 |
```

```
  3, 5 |
```

```
  5, 7 |
```

```
  4, 1 |
```

```
  2, 2 |
```

```
  0, 7 |
```

```
  8, 6 |
```

```
  |];
```

```
..... même code ....
```

👉 Les vecteurs (1d) n'auront pas besoin de ces '|'.

Les nombres réels

- On considère un problème d'emprunt à court terme (un an) à rembourser en 4 trimestres.
- On calcule les intérêts pour chaque trimestre.
- Si on emprunte 1000 à 4% et on paie 260 par trimestre, combien devrait-on à la fin de la période ?

```

var float: R; % Montant par trimestre
var float: P; % Capital principal (initial)
var 0.0 .. 10.0: I; % interest rate

% variables intermediaires
var float: B1; % balance après un trimestre
var float: B2; % balance après 2 trimestres
var float: B3; % balance après 3 trimestres
var float: B4; % Restant à la fin

constraint
  B1 = P * (1.0 + I) - R;
constraint
  B2 = B1 * (1.0 + I) - R;
constraint
  B3 = B2 * (1.0 + I) - R;

```

Les nombres réels (suite)

constraint

$B4 = B3 * (1.0 + I) - R;$

solve satisfy;

output ["Emprunter ", show_float(0, 2, P), " à", show(I*100.0),
"% interet, en payant ", show_float(0, 2, R),
"\npar semestre pour un an. Il restera ", show_float(0, 2, B4), " du\n"];

% DATA

$I = 0.04;$

$P = 1000.0;$

$R = 260.0;$

% Emprunter 1000.00 à 4.0% d'intérêt, en payant 260.00

% par semestre pour un an. Il restera 65.78 du

 Choisissez le solveur **COIN-CBC**

Réels : LP Simple

- Un simple problème d'optimisation (du livre AMPL):
 - Une compagnie fabrique 2 produits B (bands) et C (coil = bobine).
 - Le profit (par tonne) de chaque B est de 25, 30 pour C.
 - La demande de B est de maximum 6000 tonnes, 4000 pour C.
 - On produit 200 tonnes de B par heure sur les machines, 140 pour C.
 - On dispose de seulement 40 heures de production sur les machines.
- Maximiser les profits.

```

var float : quantite_B;
var float : quantite_C;
var float : Profit;

solve maximize Profit;

constraint Profit=25.0*quantite_B + 30.0*quantite_C;
constraint
  0.0 <= quantite_B           % limite B
  /\ quantite_B <= 6000.0      % limite B
  /\ 0.0 <= quantite_C        % limite C
  /\ quantite_C <= 4000.0      % limite C
  /\ (quantite_B/200.0) + (quantite_C/140.0) <= 40.0; % le temps

output ["quantite_B=", show(quantite_B), " quantite_C=", show(quantite_C), " Profit=", show
(Profit)];

```

☞ problème MIP : choisir le solveur COIN-BC dans la liste des solveurs.

Réels : LP Simple (suite)

Résultats :

$$\rightarrow XB=6000.0, \quad XC=1400.0, \quad Profit=192000.0$$

☞ Faire varier les paramètres, observer leurs effets

Remarque importante sur l'interface de Minizinc :

- On peut utiliser les capacités des processeurs multi cores pour résoudre les problèmes complexes.
- On peut également désactiver les solutions intermédiaires.
 - Demander à l'enseignant !

Notes sur Primal-Dual

Rappel : pour un programme linéaire général :

$$\begin{array}{ll} \text{Min } c^T x & \text{sur le vecteur } x \geq 0 \\ \text{s.t. } Ax \geq b. \end{array}$$

Le dual sera :

$$\begin{array}{ll} \text{Max } b^T \alpha & \text{sur le vecteur } \alpha \\ \text{s.t. } \alpha_i \geq 0 \text{ et } A^T \alpha \leq c. \end{array}$$

Symétriquement :

➔ Si dans Primal, $\text{Max}\{C^T x | Ax \leq b, x \geq 0\}$

➔ Dans Dual : $\text{Min}\{b^T \alpha | A^T \alpha \geq c, \alpha \geq 0\}$

☞ α représente le vecteur des variables de la forme Duale (on utilise souvent y comme dans les exemples).

☞ Pour les intérêts de ces deux formes, voir cours.

Notes sur Primal-Dual (suite)

Exemple : pour la forme primale

Fonction Objective : **maximize** $2*x[1]+3*x[2]$;
Les contraintes :
 $4*x[1]+8*x[2] \leq 12$;
 $2*x[1]+x[2] \leq 3$;
 $3*x[1]+2*x[2] \leq 4$;

On aura (à maximiser):

$$A = \begin{bmatrix} 4 & 8 \\ 2 & 1 \\ 3 & 2 \end{bmatrix} x \leq b = \begin{bmatrix} 12 \\ 3 \\ 4 \end{bmatrix} \text{ avec } c^T = \begin{bmatrix} 2 \\ 3 \end{bmatrix} x \text{ (maximiser)}$$

Ce qui donnera la forme Duale (à minimiser):

$$A^T = \begin{bmatrix} 4 & 2 & 3 \\ 8 & 1 & 2 \end{bmatrix} y \geq c = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \text{ avec } b^T = [12 \quad 3 \quad 4] y \text{ (minimiser)}$$

C'est à dire :

Fonction Objective : **minimize** $12*y[1]+3*y[2]+4*y[3]$;
Les contraintes :
 $4*y[1]+2*y[2]+3*y[3] \geq 2$;
 $8*y[1]+y[2]+2*y[3] \geq 3$;

Notes sur Primal-Dual (suite)

- Créer le problème (forme Primale) suivant :
 - ➔ Remarquer l'utilisation d'un vecteur simple.
- ☞ Préciser les réels constantes sinon on aura des messages d'erreur de type.

```

set of 1..2 : s = 1.. 2;
array[s] of var float : x;
var float : obj;

constraint
  forall(i in s) (x[i] >= 0.0);

constraint
  4.0*x[1]+8.0*x[2] <= 12.0 /\
  2.0*x[1]+x[2] <= 3.0 /\
  3.0*x[1]+2.0*x[2] <=4.0;

constraint
  obj = 2.0*x[1]+3.0*x[2];

solve maximize obj;

output ["x=", show(x), " objective=" , show(obj)];

%mzn-g12mip primal.mzn (ou minizinc -b mip primal.mzn)
%x=[0.49999999999999983, 1.25] objective=4.75

```

Notes sur Primal-Dual (suite)

- La forme duale du même problème :

```

set of int : J = 1.. 3;    % ou set of 1..3 : J = 1.. 3;
array[J] of var float : y;
var float : obj;

solve minimize obj;

constraint
  forall(i in J) (y[i] >= 0.0);
constraint
  obj = 12.0*y[1]+3.0*y[2]+4.0*y[3];

constraint
  4.0*y[1]+2.0*y[2]+3.0*y[3] >= 2.0
  ^
  8.0*y[1]+y[2]+2.0*y[3] >= 3.0;

output ["y=", show(y), " objective=" , show(obj)];

%mnz-g12mip dual.mzn
%y=[0.31245, 0.0, 0.25] objective=4.75

```

☞ On remarque que les *extrema* sont identiques !

Modélisation Giapetto

- L'atelier de *Giapetto* fabrique deux 2 de jouets en bois : **soldats** et **trains**.
- Un soldat se vend à 27 euros et utilise 10 euros de matières premières.
Chaque soldat coûte par ailleurs 14 euros en coûts divers (salaire, amortissement, etc.).
- Un train se vend 21 euros et utilise 9 euros en matières premières.
Chaque train coûte en frais généraux 10 euros.
- La fabrication des soldats et des trains en bois exige deux types de machines pour la menuiserie et la finition.
- Un soldat a besoin de 2 heures de finition et de 1 heure de menuiserie.
Un train a besoin de 1 heure de finition et 1 heure de menuiserie.
- L'entreprise dispose de toute la matière première. Par contre, elle ne dispose que de 100 heures de finition et 80 heures de menuiserie.
- La demande des trains est illimitée, mais au plus 40 soldats sont achetés chaque semaine.
- *Giapetto* veut maximiser ses bénéfices d'hebdomadaire.

Modélisation Giapetto (suite)

- Soit x_1 : nbr de soldats en bois produits par semaine et
 x_2 : nbr de trains produits par semaine.

- Les bénéfices seront alors

$$Z = (27 - 10 - 14)x_1 + (21 - 9 - 10)x_2 = 3x_1 + 2x_2$$

- Par ailleurs, les contraintes de production sont :

$$\begin{array}{ll} \infty \geq x_1 \geq 0 & \text{et} \quad 40 \geq x_2 \geq 0 \text{ (nbr de produits fabriqués)} \\ x_1 + x_2 \leq 80 \text{ (menuiseries)} & \text{et} \quad 2x_1 + x_2 \leq 100 \text{ (finition)} \end{array}$$

Un premier test fait en CLP :

```
Z=3*X1+2*X2 ,
X1::0..40, X2 >= 0,
X1+X2 =< 80, 2*X1+X2 =< 100,
maxof(labeling([X1,X2]),Z).

Test :
Z = 180
X1 = 20
X2 = 60
```

- Solution Minizinc

... ~>

Modélisation Giapetto (suite)

Une solution Minizinc (*giapetto.mzn*)

```
% problème giapetto
% Decision variables
var int : x1; % soldat
var int : x2 ; % train

% Objective function
solve maximize 3*x1 + 2*x2;
% Constraints
constraint
  x1 >= 0 /\ x2 >= 0
  /\ 2*x1 + x2 <= 100 % Finition
  /\ x1 + x2 <= 80 % Menuiserie
  /\ x1 <= 40; % Demande

output ["x1 = " , show(x1), " , x2 = " , show(x2), " , objective = "
  , show(3*x1 + 2*x2), "\n"];

%Exécuter minizinc giapetto.mzn
% x1 = 20, x2 = 60, objective = 180
```


Modèle et Data pour Giapetto

- Minizinc peut fonctionner sur une base plus souple de Modèle + Data.
- Le Modèle peut être figé mais le Data peut varier,
- La partie données du problème peut être insérée à la suite du modèle ou être de préférence placée dans un autre fichier.
- Pour le même problème *Giapetto*, on a le modèle Minizinc suivi de ses paramètres (dans le même fichier, comme dans l'exemple de coloration) :

```

set of int : TOY; % Pour voir les valeurs, regarder ci-dessous !

% Parameters
array[TOY] of int : Heures_Finition;
array[TOY] of int : Heures_Menuiserie;
array[TOY] of int : Demande_toys;
array[TOY] of int : Profit_toys;

% Decision variables
array[TOY] of var int : x;
var int : obj = sum(i in TOY) (Profit_toys[i]*x[i]);
%var x (i in TOY) >=0;

% Objective function
solve maximize obj;

```

Modèle et Data pour Giapetto (suite)

```

constraint
  sum(i in TOY) (Heures_Finition[i]*x[i]) <= 100; % Fin_heures

constraint
  sum(i in TOY) (Heures_Menuiserie[i]*x[i]) <= 80; % Menus_heures

constraint
  forall(i in TOY) (x[i] <= Demande_toys[i]);

output ["Obj=", show(obj), "\n"];

%-----
% la partie Data

TOY = 1..2;
Heures_Finition=[1,2];
Heures_Menuiserie=[1,1];
Demande_toys = [40, 6000000];
Profit_toys = [3,2];

%-----
% Un test : Obj=180

```

Modèle et Data pour Giapetto (suite)

- On peut donc placer la partie Data dans un autre fichier (Giapetto.dzn) :

```
% la partie Data

TOY = 1..2;

Heures_Finition=[1,2];

Heures_Menuiserie=[1,1];

Demande_toys = [40, 6000000];

Profit_toys = [3,2];

%-----
% Un test : Obj=180
```

- Si on souhaite tester le modèle sur d'autres données, on change seulement de fichier Data; le modèle ne change pas.
- Exemple tiré du document de prise en main de Glpk (Gnu Linear Programming Kit) *"Introduction to linear optimization"*, sur le site de GLPK.

Coloration plus générale

- On peut généraliser le modèle de coloration sachant que le graphe des incompatibilités est donné sous forme d'un tableau (matrice).
 - Une matrice $M : \text{noeuds} \times \text{couleur}$ contiendra le résultat final où $M[n, c]$ contiendra 1 si le noeud n reçoit la couleur c , 0 sinon.
- Avant de regarder le code, réfléchir au modèle !

```

int: nb_noeuds=5;           % nombre de noeuds

set of int: noeuds = 1..nb_noeuds; % ensemble de noeuds
int: nb_aretes=5;           % le graphe a 5 arêtes

% Le graphe :
array[1..nb_aretes, 1..2] of noeuds: aretes = [|      % remarquer les '|'
1, 5|
2, 3|
2, 4|
3, 5|
4, 5|
|];

% On sait que 4 couleurs suffisent largement
int: nb_couleurs = 4;

% matrice_noeud_couleur[i,c] = 1 veut dire : le noeud i a reçu la couleur c
array[noeuds, 1..nb_couleurs] of var 0..1: matrice_noeud_couleur;

solve satisfy;

```

Coloration plus générale (suite)

```

constraint
% CONTROLE : pas de loop dans le graphe (d'un noeud à lui même)
forall(i in 1..nb_aretes) (
    aretes[i,1] != aretes[i,2]
)
^
% Chaque noeud reçoit une seule couleur :
%      pour toute ligne de la matrice, la somme =1
forall(i in noeuds)
    (sum(c in 1..nb_couleurs) (matrice_noeud_couleur[i,c]) = 1)
^
% Deux adjacents n'auront pas la même couleur : Dans une colonne de
% la matrice, la somme des valeurs de deux noeuds voisins <= 1
forall(i in 1..nb_aretes, c in 1..nb_couleurs) (
    matrice_noeud_couleur[aretes[i,1],c] + matrice_noeud_couleur[aretes[i,2],c] <= 1
);

output ["obj: ", "\n",]
++ [" c1"++" c2"++" c3"++" c4"]
++ [if j = 1 then "\n" ++ show(i) ++ ": " else " " endif ++
    show(matrice_noeud_couleur[i,j])++" " | i in noeuds, j in 1..nb_couleurs] ++ ["\n"];
%-----
/* Solution
   c1 c2 c3 c4
1: 0 1 0 0      ← la couleur 2 donnée aux noeud 1, 3 et 4
2: 1 0 0 0      ← la couleur 1 donnée aux noeud 2 et 5
3: 0 1 0 0
4: 0 1 0 0
5: 1 0 0 0
*/

```

- Explications sur le tableau, *output*, *set*,

Coloration plus générale (suite)

Remarque sur la toute dernière contrainte sur la `matrice_noeud_couleur` (appelons la **M**) :

```

ne pas écrire   $M[arêtes[i,1],c] \neq M[arêtes[i,2],c]$ 
qui on impose que l'un des deux =1,
Or, ceci n'est pas toujours le cas (cette couleur est peut être non utilisée)
Par contre, on peut dire :
Si cette couleur est utilisée (il y a un "1" sur une des lignes de cette colonne)
Alors il faut 2 valeurs différentes pour les 2 noeuds de l'arête :
    (sum(j in 1..nb_noeuds) (matrice_noeud_couleur[j,c])) > 0
    -> % implication
    ( $M[arêtes[i,1],c] \neq M[arêtes[i,2],c]$ )

```

- Remarques sur l'implication.

Optimisation du nombre de couleurs

On reprend le code précédent pour proposer une solution qui minimise le nombre de couleurs utilisées.

- L'idée : compter le nombre de fois où une couleur est utilisée (pour au moins un noeud).
- Ajouter au code précédent la partie ci-dessous et compléter : on compte le nombre de colonnes de la matrice où au moins un "1" apparaît.

```
solve minimize nb_coul_used; % replace solve satisfy;
%....
var int : nb_coul_used = sum(c in 1..nb_couleurs) (
    bool2int(sum(v in noeuds) (matrice_noeud_couleur[v,c]) > 0)
);

% Et ajouter l'affichage de la variable nb_coul_used
```

- Dans le "morceau" de code ci-dessus, on compte, dans la matrice *noeuds* × *couleurs* le nombre de colonnes où au moins un "1" apparaît.

→ Le test `sum(v in noeuds) (matrice_noeud_couleur[v,c]) > 0` donne un résultat booléen et il faut donc le convertir en un entier (vrai=1, faux=0) via `bool2int` pour pouvoir faire une somme sur ces valeurs.

Optimisation du nombre de couleurs (suite)

Une 2e manière d'optimiser le nombre de couleurs :

- Reprendre le code précédent (sans l'optimisation ci-dessus) pour proposer une autre solution qui minimise le nombre de couleurs utilisées.
- Ajouter à ce code la partie ci-dessous et compléter : on compte le nombre de colonnes de la matrice où au moins un "1" apparaît et on stock le résultat dans un vecteur.
→ Remplacer également "solve satisfy" par le "solve" donné ci-dessus.
- Remarques sur l'implication ("→") et la méthode de calcul, minimisation.

```
var int : nb_coul_used = sum(c in 1..nb_couleurs) (vect_used_colors[c]);

solve minimize nb_coul_used;

array[1..nb_couleurs] of var int : vect_used_colors;

constraint
% Si une colonne c porte au moins un "1", mettre 1 dans vect_used_colors[c]
forall(c in 1..nb_couleurs)
(
  (((sum(i in 1..nb_noeuds) (matrice_noeud_couleur[i,c])) > 0) -> (vect_used_colors[c]
]=1))
  % Important de mettre 0
  ^ (((sum(i in 1..nb_noeuds) (matrice_noeud_couleur[i,c])) = 0) -> (vect_used_colors[c]
]=0))
);
```


Séparation Modèle-Data

- On reprend l'exemple de coloration vu plus haut.
 - Cette fois, on sépare le modèle des données (le modèle est ci-dessous)
 - Dans cette version, les données sont détachées du modèle mais placées dans le même fichier.
- Il suffira ensuite de placer la partie Data dans un autre fichier pour résoudre une autre instance de ce problème.

```
int: n;           % nombre de noeuds

set of int: V = 1..n; % set of noeuds

int: num_edges;

array[1..num_edges, 1..2] of V: E;

% 4 couleurs suffisent largement
int: nc = 4;

% x[i,c] = 1 veut dire : le noeud i prend la couleur c
array[V, 1..nc] of var 0..1: x;

solve satisfy;

constraint
```

Séparation Modèle-Data (suite)

```

% CONTROLE : pas de loop dans les donnée
forall(i in 1..num_edges) (
  E[i,1] != E[i,2]
)

^
% Tout noeud reçoit une couleur
forall(i in V) (sum(c in 1..nc) (x[i,c]) = 1)
^
% Les noeuds adjacents ne peuvent pas avoir la même couleur
forall(i in 1..num_edges, c in 1..nc) (
  x[E[i,1],c] + x[E[i,2],c] <= 1
)
;

output ["obj: ", "\n",]
++[" if j = 1 then "\n" ++ show(i) ++ ": " else " " endif ++
  show(x[i,j]) | i in V, j in 1..nc]
++ ["\n"];
%-----

```

Et les données (placées dans un fichier de données qui sera donné en paramètre du solveur ou placées après le modèle) :

Séparation Modèle-Data (suite)

```
% data

% La solution optimale : 4

n = 5;
num_edges = 5;

E = array2d(1..num_edges, 1..2, [
    1, 5,
    2, 3,
    2, 4,
    3, 5,
    4, 5
]);
```

- Il suffit donc de découper la partie Data, la placer dans (par exemple) le fichier *graphe.dzn* puis de lancer *minizinc coloration.mzn graphe1.dzn*.
→ Puis plus tard avec *graphe2.dzn*,

Exemple : où est le zèbre ?

Ce problème est énoncé par les faits suivants :

- 5 maisons consécutives dans une rue, de couleurs différentes, habitées par des hommes de nationalités différentes, chacun avec un animal différent et boit une boisson différente et fume des cigarettes de marques différentes.
- Les 5 nationalités : anglais, espagnole, ukrainien, japonais, norvégien.
- On sait que :
 - La maison verte est à droite de la maison de couleur ivoire.
 - Celui qui fume des Winstons élève des escargots.
 - On boit du lait dans la maison du milieu.
 - Le Norvégien est dans la première maison à gauche.
 - Le fumeur de Chesterfield habite à côté de celle avec un Renard.
 - Dans la maison verte, on boit du café.
 - Celui qui fume des Kools est dans la maison jaune.
 - Le fumeur de Lucky-Strike boit du jus d'orange.
 - Le Norvégien habite à côté de la maison bleue.

→ Qui boit de l'eau ? Où est le zèbre ?

.../ ...

Exemple : où est le zèbre ? (suite)

Le tableau suivant résume les informations, les complète et aide dans la résolution (raisonnement logique).

couleur	nationalité	animal	boisson	cigarettes
rouge	anglais	?	?	?
?	espagnole	chien	?	?
?	ukrainien	?	thé	?
?	japonnais	?	?	Chesterfield
?	norvégien	?	?	?

couleurs = {rouge, jaune, ivoire, verte, bleue}

animaux = {chien, renard, zèbre, escargots, cheval }

boissons = { eau minérale, lait, café, thé, jeu d'orange }

cigarettes = {Lucky-Strike, Kools, Winstons, Chesterfield, Marlboro }

Les '?' représentent les inconnues.

Qui possède le zèbre et qui boit de l'eau (minérale) ?

Solution (Zèbre)*

- Une idée : numéroter les *nationalités* de 1 à 5 :
 - ➔ Anglais=3 veut dire : le 3e maison est habitée par un anglais.
- Comment exprimer "la maison verte est à droite de la maison blanche"?
 - Une bonne idée : numéroter les maisons de 1 à 5 :
 - Et la première maison est (supposée) à gauche.
 - ➔ *Le Norvégien est dans la première maison à gauche* donne Norvégien=2
 - ➔ Jaune=2 veut dire : la 2e maison est jaune.
 - ➔ *La maison verte est à droite de la maison rouge* : Verte = Rouge +1
- ☞ On prendra un tableau de 5 entiers (1..5) pour chaque caractéristique.
- On utilise 25 variables : 5 par catégories (voir l'énoncé).
 - Le domaine de chaque variable est de 1 à 5 (les maisons).
 - Pour les animaux : on connaît chien, escargots, renard, cheval, ..
 - Pour les boissons : on a thé, café, lait, jus, ..
 - Imposer "tous différents" aux 5 paquets de 5 variables.

Solution (Zèbre)* (suite)

Le tableau précédent devient une collection de 5 tableaux chacun de 5 éléments.

couleur	nationalité	animal	boisson	cigarettes
rouge	anglais	?	?	?
?	espagnole	chien	?	?
?	ukrainien	?	thé	?
?	japonnais	?	?	Chesterfield
?	norvégien	?	?	?

- Chaque ligne : `array[1..5] of 1..5 : Couleurs;`

→ idem pour les 4 autres caractéristique.

- Pour chaque tableau, tous les éléments doivent être différents.

- Rappel des domaines des variables :

couleurs = {anglais, espagnole, ukrainien, japonais, norvégien}

couleurs = {rouge, jaune, ivoire, verte, bleue}

animaux = {chien, renard, zèbre, escargots, cheval }

boissons = {eau, lait, café, thé, jeu d'orange }

cigarettes = {Lucky-Strike, Kools, Winstons, Chesterfield, Marlboro }

Une solution Minizinc

```

include "globals.mzn";
set of 1..5: intervalle = 1..5;      % Indice 1 2 3 4 5
array[intervalle] of var intervalle: Nat; % Nat : norv esp ukr jap angl
array[intervalle] of var intervalle: Coul; % Colors: jaune bleu rouge vert ivoire
array[intervalle] of var intervalle: Anim; % Animaux : chien renard zebre escargot cheval
array[intervalle] of var intervalle: Boiss; % Boissons : eau lait cafe the orange
array[intervalle] of var intervalle: Cig; % Cigarre : lucky kools winston chester marlbo

% on décide que les maisons vont de gauche à droite et la première habité e par le Norvégien
constraint
  all_different(Nat)
  /\ all_different(Coul)
  /\ all_different(Anim)
  /\ all_different(Boiss)
  /\ all_different(Cig)
;

constraint
  forall(i in intervalle) (
    (Cig[i]=3 -> Anim[i]=4) % winston (3) => escargot (4)
    /\ % cheterfield (4) à coté du renard (2):
    (((i < 5) -> (Cig[i]=4 -> Anim[i+1]=2)) \/\ ((i > 1) -> (Cig[i]=4 -> Anim[i-1]=2)))
  );

% 2e forme d'itération (avec garde) :
constraint % vert(5) à droite ivoire(5) :
  forall([Coul[i]=5 -> Coul[i-1]=4 | i in intervalle where i < 5])
;

% 3e forme d'itération
constraint % vert(4) boit café(3)
  forall([Coul[i]=4 -> Boiss[i]=3 | i in intervalle]) ;

```


Une solution Minizinc (suite)

```

constraint
  forall(i in intervalle) (
    (Cig[i]=2 -> Coul[i]=1) % kools et jaune
    /\ (Cig[i]=1 -> Boiss[i]=5) % lucky et jus orange
  );

constraint
  Boiss[3]=2 % 3e maison (du milieu) boit du lait (2)
  /\ Nat[1]=1 % 1ere maison = norvégien
  ;

solve satisfy;

output ["National = ", show(Nat) , "\n"]
++ ["Couleur = ", show(Coul) , "\n"]
++ ["Cigarette = ", show(Cig) , "\n"]
++ ["Boisson = ", show(Boiss) , "\n"]
++ ["Animal = ", show(Anim) , "\n"]
;

/*
National = [1, 5, 4, 3, 2]
Couleur = [4, 2, 1, 3, 5]
Cigarette = [5, 3, 2, 1, 4]
Boisson = [3, 4, 2, 5, 1]
Animal = [5, 4, 3, 2, 1]
*/

```

Une solution Minizinc (suite)

Une autre solution (sans les implications) pour une variante du problème :

```
include "globals.mzn";

set of 1..5: nat = 1..5; % Nationalities: Norwegian, Dane, Briton, Swede, German
set of 1..5: col = 1..5; % Colors: yellow blue red green white
set of 1..5: pet = 1..5; % Pets: cat bird dog horse fish
set of 1..5: drink = 1..5; % Drinks: coffee tea milk juice water
set of 1..5: smoke = 1..5; % Smokes: Dunhill Marlboro Pall - Mall Blumaster Prince

array[nat] of var nat: Tnat;
array[col] of var col: Tcol;
array[pet] of var pet: Tpet;
array[drink] of var drink: Tdrink;
array[smoke] of var smoke: Tsmoke;

array[nat] of string: nationalities = ["Norwegian", "Dane", "Briton", "Swede", "German"];

solve satisfy;

constraint
  all_different(Tnat) /\
  all_different(Tcol) /\
  all_different(Tpet) /\
  all_different(Tdrink) /\
  all_different(Tsmoke) /\

  Tnat[3] = Tcol[3] /\
  Tnat[4] = Tpet[3] /\
  Tnat[2] = Tdrink[2] /\
  Tcol[4] + 1 = Tcol[5] /\
  Tcol[4] = Tdrink[1] /\
  Tsmoke[3] = Tpet[2] /\
```

Une solution Minizinc (suite)

```

Tdrink[3] = 3 ∧
Tcol[1] = Tsmoke[1] ∧
Tnat[1] = 1 ∧
(Tsmoke[2] = Tpet[1]+1 ∨ Tsmoke[2] = Tpet[1]-1) ∧
(Tpet[4] = Tsmoke[1]+1 ∨ Tpet[4] = Tsmoke[1]-1) ∧
Tsmoke[4] = Tdrink[4] ∧
Tcol[2] = 2 ∧
Tnat[5] = Tsmoke[5] ∧
(Tsmoke[2] = Tdrink[5] + 1 ∨ Tsmoke[2] = Tdrink[5] - 1)
;

output [
  "Tnat: ", show(Tnat), "\n",
  "Tcol: ", show(Tcol), "\n",
  "Tpet: ", show(Tpet), "\n",
  "Tdrink: ", show(Tdrink), "\n",
  "Tsmoke: ", show(Tsmoke), "\n"
] ++
["The " ++ show(nationalities[fix(Tnat[Tpet[5]])]) ++ " owns the fish\n"]++["\n"];

/*
Solution
Tnat: [1, 2, 3, 5, 4]
Tcol: [1, 2, 3, 4, 5]
Tpet: [1, 3, 5, 2, 4]
Tdrink: [4, 2, 3, 5, 1]
Tsmoke: [1, 2, 3, 5, 4]
The German owns the fish
*/

```

Remarques sur les exercices à rendre

- Les exercices suivants sont donnés avec leur barème (entre parenthèses).
- Respecter les consignes suivantes concernant vos choix :
 - Au moins deux exercices de 5 points (et plus) doivent être traités.
 - Au moins deux exercices dont les barèmes sont entre 3 et 4 points doivent être traités.
 - Ne pas traiter plus de 3 exercices à seulement à 1 point.
 - Le reste est laissé à votre choix.
 - Il est possible de réaliser ces exercices en Python
 - ➔ voir les packages d'optimisation sous Python tels que :
Pyomo, Pulp, CVXOPT, OPIPT, SCIP, Scipy / optimize,

👉 Si vous voulez rendre par binôme, vous devez viser 30 points !

Exercices Monnaie (1)

- Soit un *montant* (par exemple $M = 28$) et un stock de petites pièces $S = [S_1, \dots, S_k]$, $S_i \geq 1$ (par exemple, $S = [1, 7, 23]$).

On suppose que les petites pièces sont disponibles en nombres illimités.

Proposer une séquence $E = [E_1, \dots, E_k]$ où $E_j \geq 0$ représente le nombre de pièces S_j à utiliser tels que que $\sum_{j=1}^k E_j * S_j = M$.

La solution optimale \hat{E} est telle que $sum(\hat{E})$ est minimale : le nombre total de pièces utilisées est minimal.

- Par exemple, pour $M = 28$ et $S = [1, 7, 23]$, on peut proposer différentes E :
 - $E = [28, 0, 0]$ (28 fois 1, rien d'autre) : $sum(E) = 28$
 - $E = [5, 0, 1]$ (5 pièces de 1 et une pièce de 23) : $sum(E) = 6$
 - $E = [0, 4, 0]$ (4 pièces de 7) : $sum(E) = 4 \rightarrow E = \hat{E}$ est une solution **optimale**.

Exercice Paysans (3)

- 3 agriculteurs, 4 outils
- chacun donne ses préférences, un jour par outil.
- organiser en minimisant le nombre de jours
- Un outil utilisé par l'un ne peut être utilisé le même jour par un autre.
- Un agriculteur utilise chaque outil toute une journée.

Les préférences (H=taille_haies, T=tronçonneuse, B=boucheuse, M=motoculteur):

Modeste :< H, T, B, M >

Paul :< T, M, H, B >

claud :< M, T, B, H >

- Une solution possible :

	H	M	B	T
M	j1	j4	j3	j2
P	j3	j2	j5	j1
C	j5	j1	j4	j3

Exercice BIP (trivial, 1)

- Donner des conseils à cet entrepreneur pour un problème de type entrepôt (Modélisation BIP) :

Une entreprise souhaite construire une nouvelle usine à Lyon ou à Grenoble (ou les deux).

- Un seul entrepôt mais là où on aura construit l'usine.
- Le bénéfice et le cout de chaque est donné dans la table ci-dessous.
- L'objectif est de maximiser les bénéfices.

Question oui/non	variable de décision	Bénéfices	couts
usine à Lyon	X_1	9 millions	6 millions
usine à Grenoble	X_2	5	3
entrepôt à Lyon	X_3	6	5
entrepôt à Grenoble	X_4	4	2

Capitaux disponibles max

10 millions.

- Combien faut-il des capitaux pour un rapport maximal $\frac{\text{gains}}{\text{couts}}$

Exercice : Gâteaux (3)

- Un pâtissier a deux recettes :

Tarte à la banane

250g de farine
2 bananes,
75g sucre
100g beurre

Tarte au chocolat

200g de farine
75g cacao,
150g sucre
150g beurre

- Il dispose de 4kg de farine, 6 bananes, 2kg de sucre, 500g de beurre et 500g de cacao.
- Il vend les tartes de chocolat à **4,5 euros** et 4,0 euros pour les tartes à la banane.
- Maximiser ses bénéfices.

Exercice : Régression (3)

- On dispose de quelques points et on souhaite procéder à une régression linéaire pour ces points. La droite représentative doit minimiser l'erreur absolue (mais linéaire).
- Proposer une solution permettant de trouver les coefficients de la droite $y = f(x) = ax + b$ et donner l'erreur.
- Les points x_i, y_i dans $f(x) = y$ sont (p.ex. pour $n = 13$ points):

1.0,	3.6,
1.5,	3.7,
2.0,	3.9,
2.5,	4.3,
3.0,	4.5,
3.5,	4.72,
4.0,	5.1,
4.2,	5.2,
5.0,	5.7,
5.8,	6.0,
6.0,	6.2,
6.7,	6.4,
7.2,	6.7

- Indication 1 : minimiser une somme d'erreurs linéaires car les contraintes quadratique (donc non linéaires) ne sont pas prises en charge par *Minzinc*.

Exercice : Régression (3) (suite)

- Indication 2 : la fonction $abs(.)$ s'appliquera ici que si un domaine est précisé pour a, b (on peut prendre p.ex. $[-100.0.. +100.0]$).

☞ Au lieu d'écrire

$$err_i \leq abs(ax_i + b - y_i)$$

où a, b sont les coefficient de la droite

à trouver, on peut plutôt utiliser :

$$err_i \geq ax_i + b - y_i \text{ ET } err_i \geq -(ax_i + b - y_i)$$

- Indication 3 : pour un cas à 2 dimensions, l'expression des coefficients a et b par la méthode moindres carrés (minimisation des carrés des erreurs) permet une meilleure approximation :

$$\hat{a} = \frac{n \cdot \sum_1^n x_i y_i - \sum_1^n x_i \cdot \sum_1^n y_i}{n \cdot \sum_1^n (x_i)^2 - \left(\sum_1^n x_i \right)^2}$$

$$\hat{b} = \frac{\sum_1^n (x_i)^2 \cdot \sum_1^n y_i - \sum_1^n x_i \cdot \sum_1^n x_i y_i}{n \cdot \sum_1^n (x_i)^2 - \left(\sum_1^n x_i \right)^2}$$

Exercice : Régression (3) (suite)

- Comme il a été indiqué en cours, si vous ne disposez pas de la fonction $abs(.)$ (ce qui n'est pas notre cas ici !), si $abs(.)$ ne s'applique pas aux réels ou enfin si vous devez optimiser son utilisation, vous pouvez utiliser la technique suivante pour la transformer.

L'expression qui peut remplacer $abs(.)$:

- Soit la contrainte $abs(f(x)) \geq m$
- On pose (z est un booléen, U et L sont les bornes Sup/Inf de m) :
 - Si $z = 1$ alors on a $f(x) \geq 0$ et donc $U \geq f(x) \geq m$ (I)
 - Si $z = 0$ alors on a $f(x) < 0$ et donc $-m \geq f(x) \geq L$ (II)
 - Et donc : $z * (U + m) - m \geq f(x) \geq L + z * (m - L)$ (III)
- Dans le cas présent, $m = err_i$ est l'erreur individuelle pour x_i
 et $f(x) = ax + b - y$.
- On peut raisonnablement poser $L = 0.0$ et $U = 1.0$
 (mais une valeur plus grande pour U p.ex. 100 est possible, cf. Big_M)
- Dans ce cas, (III) devient $m \geq f(x) \geq -m$

Exercice voyage (2)

Exercice Tourné (un exemple BIP)

- Pour une prospection, un étudiant veut visiter les campus de trois universités en Rhône-Alpes pendant un voyage à partir de "Lyon" et retour.

Les trois Univ sont situées dans les villes "St-Etienne", "Valence" et "Grenoble" et l'étudiant veut visiter chaque ville universitaire une seule fois tout en faisant l'aller-retour **le plus court possible**.

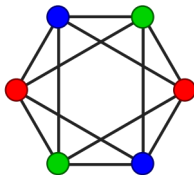
- La table suivante donne les distances entre les villes :

Villes	Ville 1	Ville 2	Ville 3	Ville 4
	Lyon	St-Etienne	Valence	Grenoble
Lyon	0	26	34	78
St-Etienne	26	0	18	52
Valence	34	18	0	51
Grenoble	78	52	51	0

- Proposer une solution Minizinc.

Exercice coloration (2)

- Soit le graphe suivant à colorer de sorte que 2 noeuds connectés ne reçoivent pas la même couleur.



- Proposer une solution Minizinc minimisant le nombre de couleurs utilisées.

Exercice fret (1)

Minimisation du cout total.

Transporter 42 tonnes de fret avec 8 camions de 4 volumes différents :

Type	Nb dispo	Capacité (tonnes)	cout par camion
1	3	7	90
2	3	5	60
3	3	4	50
4	3	3	40

Modèle :

- Variables : $x_{i=1..4}$: nombre de chaque camion utilisé (i=le type : 1..4)

$$\min \quad 90x_1 + 60x_2 + 50x_3 + 40x_4$$

$$s.t. \quad 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42$$

$$x_1 + x_2 + x_3 + x_4 \leq 8$$

$$7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 \quad \text{appelé "contrainte couverture sac-à-dos"}$$

$$x_1 + x_2 + x_3 + x_4 \leq 8 \quad \text{appelé "contrainte remplissage sac-à-dos"}$$

- Proposer une solution Minizinc.

Exercice Jobshop (5)

- Une série de tâches avec pour chacune une date au plus tôt ($Rel=release$) où la tâche peut commencer, une durée de la tâche et une date de livraison (qu'on va dépasser pour certaines).
- Si la date de livraison d'une tâche est dépassée, on tient compte de la différence (pénalité en vue!); par contre, si une tâche est prête avant sa date de livraison, le dépassement = 0 (car il nous faut tout finir pour livrer).
- On a une seule machine : donc tâche après tâche passent sur celle-ci.
→ Minimiser les dépassement.
- Dans tous les cas, la somme des durées (30) dépasse la date de livraison (22). On nous accordera peut être la différence (8), mais pas plus!
- Les données du problème sont ci-dessous :

Exercice Jobshop (5) (suite)

rel : Le moment où une tâche est disponible pour la machine
dur : Durée de la tâche
due : Le moment où la tâche doit être terminée

<i>JOBS</i>	<i>rel</i>	<i>dur</i>	<i>due</i>
A	2	5	10
B	5	6	21
C	4	8	15
D	0	4	10
E	0	2	5
F	8	3	15
G	9	2	22

/ Une solution à trouver :*

<i>Task</i>	<i>Rel</i>	<i>Dur</i>	<i>Due</i>	<i>Start</i>	<i>Finish</i>	<i>Pastdue (dépassement)</i>
A	2	5	10	2	7	0
B	5	6	21	14	20	0
C	4	8	15	22	30	15
D	0	4	10	7	11	1
E	0	2	5	0	2	0
F	8	3	15	11	14	0
G	9	2	22	20	22	0

**/*

Exercice : Jésuites (5)

Trouver une solution pour le problème suivant.

Énoncé :

- Il y a 7 jésuites dans une maison (abbaye)
- Leurs taches sont divisées en 4 groupes :
Cuisine, SdB, Partie commune et Poubelles.
- La poubelle est la seule tâche qui a besoin d'une personne;
les autres tâches ont besoin de 2 personnes.
- Chaque personne a besoin de faire une tache 2 fois sur les 7 semaines (et celle de la poubelle une fois)
- Trouver une affectation par semaine (sur les 7 semaines)

Exercice : Jésuites (5) (suite)

Modélisation :

- Toute personne doit avoir un nouveau partenaire chaque semaine (donc ne pas reconduire une équipe de 2 d'une semaine à l'autre)
- Personne ne doit avoir plus d'une tâche par semaine
- On utilise un tableau à 4-1 dimensions de bool avec S : les semaines, T : tâches, V : volontaires (les jours J : pas besoin !)
 $Cube[S : 1..7, T : 1..4, V : 1..7] :: 0..1$
- $Cube[S, T, V] = 1$ veut dire que le volontaire V fera la tâche T la semaine S
 → Créer cette cube dimension par dimension pour pouvoir faire facilement des sommes!
- Une personnes pour la poubelle, 2 pour les autres.
- Faire la somme des ligne dans chaque semaine.

../..

Exercice : Jésuites (5) (suite)

Les contraintes :

Pour tout $S : 1..7$

pour tout $T :$

si $T = 4 \rightarrow \text{somme}(\text{Cube}[S,T])=1$ Poubelle

si $T \neq 4 \rightarrow \text{somme}(\text{Cube}[S,T])=2$ Autres

- Toute personne doit faire une tâche 2 fois (sur les 7 semaines) :

Pour tout $V : 1..7$

pour tout $T :$

pour tout $S, S', J \mid S \neq S'$

si $T = 4 \rightarrow \text{Cube}[S,T,V] + \text{Cube}[S',T,V]=1$ Poubelle

si $T \neq 4 \rightarrow \text{Cube}[S,J,V] + \text{Cube}[S',T,V]=2$ Autres

- Chaque personne doit avoir un nouveau partenaire chaque semaine (donc ne pas reconduire une équipe de 2 d'une semaine à l'autre)

d'où $S \neq S'$

- Personne ne doit avoir plus d'une tâche par semaine

Rappel : $\text{Cube}[S, T, V] = 1$ veut dire que le volontaire V fera la tâche T la semaine S

Pour tout S somme de chaque colonne = 1

Exercice : Jésuites (5) (suite)

Exemple de solution :

```
Semaine : 1
Volontaire:1 2 3 4 5 6 7
Kitchen : - - - - X X
Bathroom: - - - X X -
Commons : X X - - - -
Trash : - - X - - -
```

```
Semaine : 2
Volontaire:1 2 3 4 5 6 7
Kitchen : - X X - - -
Bathroom: - - - X X -
Commons : - - - X - X
Trash : X - - - - -
```

```
Semaine : 3
Volontaire:1 2 3 4 5 6 7
Kitchen : X - - X - -
Bathroom: - - X - - X
Commons : - X - X - -
Trash : - - - - X -
```

```
Semaine : 4
Volontaire:1 2 3 4 5 6 7
Kitchen : - - X X - -
Bathroom: - X - - - X
Commons : X - - - X -
Trash : - - - - X -
```

```
Semaine : 5
Volontaire:1 2 3 4 5 6 7
Kitchen : - X - - X -
```

Exercice : Jésuites (5) (suite)

```
Bathroom: X - - X - - -
Commons  : - - X - - X -
Trash    : - - - - - X
```

Semaine : 6

```
Volontaire:1 2 3 4 5 6 7
Kitchen    : - - - X - X -
Bathroom   : X - X - - - -
Commons    : - - - - X - X
Trash      : - X - - - - -
```

Semaine : 7

```
Volontaire:1 2 3 4 5 6 7
Kitchen    : X - - - - X
Bathroom   : - X - - - X -
Commons    : - - X - X - -
```

Exercice : Jésuites (5) (suite)

Et par paires :

Pair: (1,2)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -
 Bathroom: - - - - -
 Commons : X - - - -
 Trash : - - - - -

Pair: (1,3)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -
 Bathroom: - - - - X -
 Commons : - - - - -
 Trash : - - - - -

Pair: (1,4)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -
 Bathroom: - - - X - -
 Commons : - - - - -
 Trash : - - - - -

Pair: (1,5)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - X - - -
 Bathroom: - - - - -
 Commons : - - - - -
 Trash : - - - - -

Pair: (1,6)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -

Exercice : Jésuites (5) (suite)

Bathroom : - - - - -
Commons : - - - X - - -
Trash : - - - - -

Pair : (1,7)

Semaine : 1 2 3 4 5 6 7
Kitchen : - - - - - X
Bathroom : - - - - -
Commons : - - - - -
Trash : - - - - -

Pair : (2,3)

Semaine : 1 2 3 4 5 6 7
Kitchen : - X - - - -
Bathroom : - - - - -
Commons : - - - - -
Trash : - - - - -

Pair : (2,4)

Semaine : 1 2 3 4 5 6 7
Kitchen : - - - - -
Bathroom : - - - - -
Commons : - - X - - -
Trash : - - - - -

Pair : (2,5)

Semaine : 1 2 3 4 5 6 7
Kitchen : - - - - X - -
Bathroom : - - - - -
Commons : - - - - -
Trash : - - - - -

Pair : (2,6)

Semaine : 1 2 3 4 5 6 7

Exercice : Jésuites (5) (suite)

Kitchen : - - - - -
Bathroom : - - - - - X
Commons : - - - - -
Trash : - - - - -

Pair: (2,7)
Semaine : 1 2 3 4 5 6 7
Kitchen : - - - - -
Bathroom : - - - X - -
Commons : - - - - -
Trash : - - - - -

Pair: (3,4)
Semaine : 1 2 3 4 5 6 7
Kitchen : - - - X - -
Bathroom : - - - - -
Commons : - - - - -
Trash : - - - - -

Pair: (3,5)
Semaine : 1 2 3 4 5 6 7
Kitchen : - - - - -
Bathroom : - - - - -
Commons : - - - - - X
Trash : - - - - -

Pair: (3,6)
Semaine : 1 2 3 4 5 6 7
Kitchen : - - - - -
Bathroom : - - - - -
Commons : - - - - X -
Trash : - - - - -

Exercice : Jésuites (5) (suite)

Pair: (3,7)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: - - X - - -

Commons : - - - - -

Trash : - - - - -

Pair: (4,5)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: X - - - - -

Commons : - - - - -

Trash : - - - - -

Pair: (4,6)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - - X -

Bathroom: - - - - -

Commons : - - - - -

Trash : - - - - -

Pair: (4,7)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: - - - - -

Commons : - X - - - -

Trash : - - - - -

Pair: (5,6)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: - X - - - -

Commons : - - - - -

Exercice : Jésuites (5) (suite)

Trash : - - - - -

Pair: (5,7)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: - - - - -

Commons : - - - - - X -

Trash : - - - - -

Pair: (6,7)

Semaine : 1 2 3 4 5 6 7

Kitchen : X - - - - -

Bathroom: - - - - -

Commons : - - - - -

Trash : - - - - -

Exercice : Ateliers véhicules (5)

Planification de chaîne de production de 4 ateliers A, B, C, D

- A a une capacité de production de 5 véhicules/heure/homme
- B a une capacité de production de 3 V/h/h
- C a une capacité de production de 2 V/h/h
- D a une capacité de production de 3 V/h/h
- ➔ Tout véhicule a besoin de passer devant les 4 ateliers (dans cet ordre).
- Les ressources : on a 4 opérateurs Operateur1 .. Operateur4
 - Operateur1 est opérationnel entre 8h-13h
 - Operateur2 est opérationnel entre 8h-13h
 - Operateur3 est opérationnel entre 9h-13h
 - Operateur4 est opérationnel entre 10h-15h
- A 8h, 30 véhicules sont en attente devant l'atelier A, aucun devant les autres
- 👉 **Variantes** : des véhicules de modèles différents, avec options différentes
Ateliers / Opérateurs mono / multi compétences, ...

Exercice : Ateliers véhicules (5) (suite)

- *Proposer un emploi du temps heure par heure précisant l'affectation des ouvriers aux ateliers tout en **maximisant** le total de véhicules passés devant les 4 ateliers.*

Exercice : Découpe papier (2)

Du livre *Understading and using Linear programming*, page 26.

- Trouver une solution pour le problème suivant.
- A paper mill manufactures rolls of paper of a standard width 3 meters. But customers want to buy paper rolls of shorter width, and the mill has to cut such rolls from the 3 m rolls. One 3 m roll can be cut, for instance, into two rolls 93 cm wide, one roll of width 108 cm, and a rest of 6 cm (which goes to waste). Let us consider an order of
 - 97 rolls of width 135 cm,
 - 610 rolls of width 108 cm,
 - 395 rolls of width 93 cm, and
 - 211 rolls of width 42 cm.

What is the smallest number of 3 m rolls that have to be cut in order to satisfy this order, and how should they be cut?

Exercice Tournée (contrainte element, 2)

Organisation de tournée.

- Dans un problème de tournée, un contrôleur visite des sites :
 - Il visite le site S le jour J .
 - Il ne visite chaque site qu'une fois;
 - Il peut visiter 2 sites consécutifs en 2 jours consécutifs ou espacés d'au plus 1 jour.
- **Modélisation :**
 - Variables : site S_i , l'ensemble donne $Z = \text{Liste des sites ordonnée}$
 - Domaine : $S_i \in 1..7$ (les 7 jours).
 - Contraintes :
 - Visiter chaque site une seule fois : $\text{alldifferent}(Z)$
 - Espacement des visites : pour toute paire de sites S_i et S_j consécutifs :

$$S_i = S_j + X, I \in 1..4, \text{element}(I, [-1, -2, 1, 2], X) .$$
- ☞ Quid si : si lundi travaillé, repos mercredi (et vice versa)

Exercice Bin Packing (3)

- Étant donné un ensemble d'articles $I = \{1, \dots, m\}$ avec un poids $w[i] > 0$, le problème d'emballage de bac (Bin Packing, BPP) est d'emballer les articles dans des bacs de c capacité de telle façon que le nombre de bacs utilisés soit minimal.
- Par exemple :
 - Soit des petites boites de poids $< 1, 2, 3, 4, 5, 6, 7 >$
 - Soit des grandes boites de capacité 20.
 - Emballer les petite boites en minimisant le nombre de grandes boites utilisées.
- Donner l'esquisse d'un modèle général et transformer votre programme en modèle + data (si ce n'est pas le cas).
 - Puis emballer 3 séries de petites boites. Minimiser les grandes !
- ☞ Il s'agit d'envisager un seul emballage des ces 3 séries en un seul envoi.

Bonus : Tournoi à organiser (5)

Un organisateur sportif prévoit un tournoi avec 8 équipes [Helmut Simonis-2009] :

- chaque équipe joue contre toutes les autres équipes une seule fois.
 - le tournoi se joue sur 7 jours,
 - chaque équipe joue tous les jours (des 7),
 - les matchs sont prévus sur 7 sites, et
 - chaque équipe doit jouer dans chaque site exactement une fois.
- Dans le cadre d'un accord avec la télévision, certains matchs sont pré-organisés.
- ➔ On peut soit fixer le match entre deux équipes particulières à une date déterminée et un site déterminé,
 - ➔ Ou seulement décider par avance qu'une certaine équipe doit jouer un jour donné en un lieu donné.
- **L'objectif** est de déterminer le calendrier, de sorte que toutes les contraintes soient satisfaites.

Bonus : Tournoi à organiser (5) (suite)

Expressions des exigences (contraintes) :

→ permettent d'envisager différentes contraintes / solutions :

- Il y a 8 équipes, 7 jours et 7 sites
- Chaque équipe joue chaque autre équipe une fois exactement
- Chaque équipe joue 7 matchs (redondant)
- Chaque équipe joue exactement une fois dans chaque site
- Chaque équipe joue chaque jour exactement une fois
- Un match se compose de 2 différentes équipes
- Il y a 4 matchs chaque jour (redondant)
- Il y a 4 matchs à chaque emplacement (redondants)
- Dans n'importe quel site, il y a au plus un match à la fois

☞ Selon les choix, on peut proposer différentes idées de solution ../..

Bonus : Tournoi à organiser (5) (suite)

- **Idée 1 :** utiliser une matrice $Jour \times match$ (7×4)
 - Chaque cellule contient 2 variables (deux équipes)
 - Contrainte : toute équipe joue une seule fois chaque jour (*alldifferent*)
 - Les colonnes n'auront pas de signification
 - Les sites non représentés : comment faire ?
 - Comment dire : chaque équipe joue avec une autre une seule fois.
- **Idée 2 :** variables binaires pour exprimer : équipe i joue en site j le jour k .
 - Matrice à 3 dimensions
 - Chaque équipe joue une seule fois chaque jour
 - Chaque équipe joue une seule fois dans chaque site
 - Un match a 2 différentes équipes ? (variables auxiliaires nécessaires).
 - Chaque pair d'équipe se rencontrent une seule fois ? (vars auxiliaires).
- **Idée 3 :** variables booléennes pour exprimer : équipe i rencontre équipe j en site k le jour l .
 - $3136 = 8 * 8 * 7 * 7$ variables
 - Toutes les contraintes sont linéaires

Bonus : Tournoi à organiser (5) (suite)

- Idée 4 : chaque équipe joue contre une autre exactement une fois :
 - $7 * 4$ matchs à organiser (28 variables)
 - Toutes les variables différentes (pas de match le même jour, le même lieu)
 - Par construction, chaque équipe jouera 7 matchs
 - Comment dire : chaque équipe jouera une fois par jour ?
 - Comment dire : chaque équipe jouera une fois par site ?
- Cette idée donnera :

	City 1	City 2	City 3	City 4	City 5	City 6	City 7
Day 1	1	2	3	4	5	6	7
Day 2	8	9	10	11	12	13	14
Day 3	15	16	17	18	19	20	21
Day 4	22	23	24	25	26	27	28
Day 5	29	30	31	32	33	34	35
Day 6	36	37	38	39	40	41	42
Day 7	43	44	45	46	47	48	49

Bonus : Tournoi à organiser (5) (suite)

- Jour 1 : valeurs 1..7
- 4 variables prendront une des ces valeurs (1..7)
- Jour 2 : 8..15, &c.
- Une contrainte par jour
- Exactement 4 variables prendront leur valeurs dans la ligne correspondantes
- 7 de ces contraintes (car 7 lignes).

- Le site 1 correspond aux valeurs 1,8,15, 22, ...
- 4 variables prendront une des ces valeurs
- Site 2 correspond à 2, 9,16, ...
- Une contrainte par site
- Exactement 4 variables prendront leur valeurs dans l'ensemble correspondant
- 7 de ces contraintes sur chacune des 28 variables.

- Choisir les variables qui correspondent à l'équipe i
 - Exactement une de ces variables prendront un evaleur dans 1..7
- De même pour les autres jours
- De même pour les autres matches
 - 56 contraintes sur chacune des 7 variables
- De même pour les équipes et sites :
 - 56 autres contraintes

	City 1	City 2	City 3	City 4	City 5	City 6	City 7
Day 1	1	2	3	4	5	6	7
Day 2	8	9	10	11	12	13	14
Day 3	15	16	17	18	19	20	21
Day 4	22	23	24	25	26	27	28
Day 5	29	30	31	32	33	34	35
Day 6	36	37	38	39	40	41	42
Day 7	43	44	45	46	47	48	49

Bilan de l'idée 4 :

- 28 variables,
- Un *alldifferent*
- 7 contraintes sur toutes les variables (pour les jours)
- 7 contraintes sur toutes les variables (pour les sites)
- 56 contraintes sur 7 variables (pour les jours)
- 56 contraintes sur 7 variables (pour les sites)
- ...et on n'a pas encore fini!

Bonus : Tournoi à organiser (5) (suite)

- Idée 5 (reprise en CP) :
 - une matrice carré de $Jour \times Site = 49$ de matchs (couples d'équipes $[A,B]$)
 - Chaque cellule contient un match (couples d'équipes)
 - Comment éviter les symétries ($[A,B]$ vs $[B,A]$)
 - Utiliser $[0,0]$ pour l'absence de match
 - Une cellule contient $[0,0]$ ou $[A,B]$, $A \neq B \neq 0$.
 - Plus facile : chaque ligne / colonne contient un match une seule fois.
 - Attention à *alldifferent* (pour les zéros)
 - Comment exprimer : chaque paire ordonnée apparaît une seule fois ?
 - Comment exprimer : chaque équipe joue une fois par jour / site ?
- Et du côté des contraintes non-domaine-fini ?
- 👉 Mettre mon ex de choix SD n-reines / complexité

Une solution CP

- Le tableau initial (avec les matchs pré organisés), puis une solution :

	City 1	City 2	City 3	City 4	City 5	City 6	City 7
Day 1		8			7, 5		
Day 2	2	1, 5					
Day 3	7		8				
Day 4					2	5	1
Day 5	8					1	
Day 6				5, 4			
Day 7	4				1, 3		

	City 1	City 2	City 3	City 4	City 5	City 6	City 7
Day 1		6, 8		1, 2	5, 7		3, 4
Day 2	2, 3	1, 5			4, 8	6, 7	
Day 3	1, 7	2, 4	3, 8				5, 6
Day 4			4, 7		2, 6	3, 5	1, 8
Day 5	5, 8			3, 6		1, 4	2, 7
Day 6		3, 7	1, 6	4, 5		2, 8	
Day 7	4, 6		2, 5	7, 8	1, 3		

Une solution CP (suite)

Une spécification (applicable à cette classe de problèmes) :

Placez numéros 1 à 8 dans les cellules de la matrice de sorte que :

- dans chaque rangée et dans chaque colonne figure chaque numéro d'équipe ($\in 1..8$) exactement une fois, et
 - chaque cellule contient soit pas de chiffres, soit un couple de chiffres ($\in 1..8$, = un match) différents, et
 - chaque couple de chiffres (un match) apparaît dans seulement une cellule.
- On peut envisager la structure de données suivante :
 Pour 8 équipes ($E1..E8$) :
 - M : une matrice 7×7
 - Lignes : les jours ($Ji : i = 1..7$)
 - Colonnes : les Sites ($Si, i = 1..7$)
 - Une case : vide ou $[Eu, Ev]$: les deux équipes qui jouent

Une solution CP (suite)

- **Choix des variables :**

Une matrice carré \mathbf{M} de $7 \times 7 = 49$ de couples d'équipes (*Jours* \times *Site*)

→ Chaque couple dans une cellule $M_{ij}, i, j = 1..7$

- **Domaines :**

49 cellules contenant chacune un couple $[Eu, Ev], u, v \in 0..8, u \neq v$,

☞ Pour une cellule vide (pas de match *jour* \times *site*), on utilisera le couple $[0, 0]$.

- **Contraintes :**

Pour simplifier et éviter les solutions symétriques, on décide que dans un couple non vide (donc : $\neq [0, 0]$) $[Eu, Ev] : u < v$.

- Cel_i est autorisée à contenir soit $[0, 0]$ soit $[Eu, Ev], Eu \neq 0, Ev \neq 0$
- Déf : $[Eu, Ev] \neq [Eu', Ev']$ si $u \neq u'$ ou $v \neq v'$
- Pour chaque ligne de la matrice M contenant 7 cellules $Cel_{i=1..7}$ (représentant tous les matchs d'un jour) :
 - Si $Cel_i = [Eu, Ev] \neq [0, 0]$ alors $Cel_i \neq Cel_{k=1..7, i \neq k}$
 - Que toutes les équipes soient impliquées dans $Cel_{i=1..7}$

Une solution CP (suite)

- Pour chaque colonne de la matrice M contenant 7 cellules $Cel_{j=1..7}$ (représentant tous les matchs d'un site) :
 - Si $Cel_j = [Eu, Ev] \neq [0, 0]$ alors $Cel_j \neq Cel_{k=1..7, k \neq j}$
 - Que toutes les équipes soient impliquées dans $Cel_{j=1..7}$

- Il y a beaucoup de solutions.

- Une solution (autre) :

$[[0, 0], [0, 0], [0, 0], [1, 2], [3, 4], [5, 6], [7, 8]]$

$[[7, 8], [3, 4], [5, 6], [0, 0], [0, 0], [0, 0], [1, 2]]$

$[[5, 6], [0, 0], [0, 0], [3, 4], [8, 7], [1, 2], [0, 0]]$

$[[0, 0], [7, 2], [1, 8], [0, 0], [0, 0], [3, 4], [5, 6]]$

$[[3, 4], [6, 8], [0, 0], [5, 7], [1, 2], [0, 0], [0, 0]]$

$[[0, 0], [1, 5], [2, 7], [6, 8], [0, 0], [0, 0], [3, 4]]$

$[[1, 2], [0, 0], [3, 4], [0, 0], [5, 6], [7, 8], [0, 0]]$

- Aspects déclaratifs de la CP.

PERT avec ressources : 5 (2 si cumulative utilisée)

- Minimiser le temps total (*makespan*) de réalisation des tâches sans deadline.
- On a :
 - 5 tâches t_1, \dots, t_5
avec les durées respectives 3, 3, 3, 5, 5
et les ressources nécessaires requises 3, 3, 3, 2, 2
- ☞ Minimiser la date de la fin des travaux.
- Il n'y a pas de contrainte de précédence.
- Une modélisation CP est donnée en page suivante.

Indication : pour la gestion des ressources limitées, on peut envisager (au besoin) de contrôler chaque unité du temps.

PERT avec ressources : 5 (2 si cumulative utilisée) (suite)

Une modélisation CP (avec *cumulative*) :

$\min Z$

s.t. $\text{cumulative}((t_1, \dots, t_5),$
 $(3, 3, 3, 5, 5), (3, 3, 3, 2, 2), 7)$

$Z \geq t_1 + 3$

.....

$Z \geq t_5 + 2$

Avec :

(t_1, \dots, t_5) : date de début des tâches

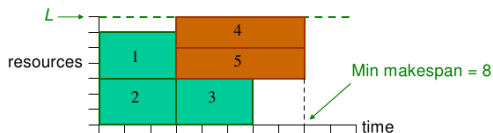
$(3, 3, 3, 5, 5)$: durées des tâches

$(3, 3, 3, 2, 2)$ ressources utilisées par les tâches

→ Ne jamais dépasser la limite des ressources 7

→ On a le total des durées = 8 (vs la somme des durées = 19)

☞ *cumulative* existe en Minizinc.



Sur les docks (5)

Du manuel OPL (un langage d'optimisation) :

Les données du problème :

- Planifier 34 containers sur un bateau en un temps minimum (min *makespan*).
 - Le chargement de chaque container nécessite un certain temps et un certain nombre d'ouvriers (cf. tableau).
 - On dispose de 8 ouvriers.
- ☞ **Pour simplifier** : considérer seulement une partie de cette table.

Container	Durée	Nb. Ouvriers
1	3	4
2	4	4
3	4	3
4	6	4
5	5	5
6	2	5
7	3	4
8	4	3
9	3	4
10	2	8
11	3	4
12	2	5
13	1	4
14	5	3
15	2	3
16	3	3
17	2	6

Container	Durée	Nb. Ouvriers
18	2	7
19	1	4
20	1	4
21	1	4
22	2	4
23	4	7
24	5	8
25	2	8
26	1	3
27	1	3
28	2	6
29	1	8
30	3	3
31	2	3
32	1	3
33	2	3
34	2	3

Sur les docks (5) (suite)

- Les contraintes de précédence :

1 → 2,4	11 → 13	22 → 23
2 → 3	12 → 13	23 → 24
3 → 5,7	13 → 15,16	24 → 25
4 → 5	14 → 15	25 → 26,30,31,32
5 → 6	15 → 18	26 → 27
6 → 8	16 → 17	27 → 28
7 → 8	17 → 18	28 → 29
8 → 9	18 → 19	30 → 28
9 → 10	18 → 20,21	31 → 28
9 → 14	19 → 23	32 → 33
10 → 11	20 → 23	33 → 34
10 → 12	21 → 22	

$t_i \rightarrow t_j$: t_j commence après la fin de t_i .

- Le modèle CP :

$\min \quad Z$

$s.t. \quad \text{cumulative}((t_1, \dots, t_{34}),$
 $(3, 4, 2, \dots, 2), (4, 4, \dots, 3), 8)$

$Z \geq t_1 + 3$

$Z \geq t_2 + 4$

.....

$t_2 \geq t_1 + 3$

$t_4 \geq t_1 + 3$

.....

Infirmières (5)

- Soit 4 infirmières travaillant pendant des périodes (shift) de 8 heures.
- Une infirmière travaille au plus un shift par jour.
- Une infirmière travaille au moins 5 jours la semaine.
- Même planification pour chaque semaine.
- Chaque shift contient deux infirmières (différentes) par semaine.
- Une infirmière ne peut pas travailler 2 shifts différents sur 2 jours consécutifs.
- Une infirmière qui travaille shift 2 ou 3 doit faire de même au moins deux jours d'affiler.
- Exemple d'assignation des infirmières aux shifts (créneaux longs).

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Shift 1	A	B	A	A	A	A	A
Shift 2	C	C	C	B	B	B	B
Shift 3	D	D	D	D	C	C	D

- Exemple d'assignation des shifts aux infirmières.

Infirmières (5) (suite)

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Nurse A	1	0	1	1	1	1	1
Nurse B	0	1	0	2	2	2	2
Nurse C	2	2	2	0	3	3	0
Nurse D	3	3	3	3	0	0	3

☞ Donner un tableau des shifts en respectant les contraintes.

→ Il y a plusieurs solutions possibles.

Restaurant (5)

- Résoudre le problème du restaurateur :
 - il veut minimiser son nombre d'employés sachant qu'un employé travaille 5 jours d'affilée puis a deux jours de repos.
 - Le personnel requis par jour de la semaine est donné par le tableau :

Jour	L	M	M	J	V	S	D
Nombre	14	13	15	16	19	18	11

- Vous donnerez le nombre total d'employés et la répartition par jour de travail.

Financement (3)

Résoudre le problème de financement ci-dessous :

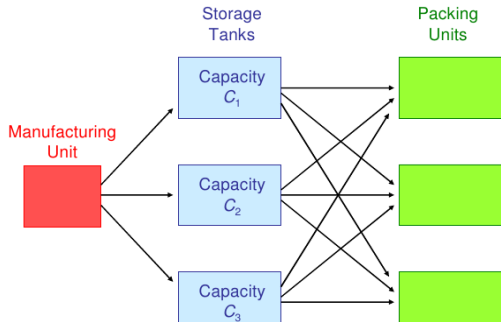
- Placer 1000 euros sur 6 ans de la manière optimale sachant que :
 - la caisse d'épargne rapporte 5% par an et immobilise le capital un an,
 - l'obligation 1 rapporte 12% à l'échéance si on le choisit la première année sinon elle rapporte 11% et immobilise le capital deux ans,
 - l'obligation 2 rapporte 18% à l'échéance et immobilise le capital trois ans,
 - l'obligation 3 rapporte 24% à l'échéance et immobilise le capital quatre ans,
 - L'obligation 2 est disponible tous les ans sauf l'année 3,
 - l'obligation 2 n'est pas disponible l'année 1,
 - et l'obligation 3 n'est disponible que l'année 1.
- Vous donnerez le gain attendu et la répartition par type de placement chaque année
- Calculez le taux moyen de rendement et comparez-le au taux de la caisse d'épargne.

Voyageur (3)

- Un voyageur peut rapporter pour les vendre divers objets.
 - Malheureusement la compagnie aérienne limite la charge qu'il peut emporter dans son sac à dos à 40kg ce qui lui laisse uniquement 14kg (car il a déjà ses propres affaires).
 - Le premier objet pèse 5kg et se revend avec un bénéfice de 8 euros,
 - Le second pèse 7kg et rapporte 11 euros,
 - Le troisième pèse 4kg et rapporte 6 euros
 - Et le dernier pèse 3kg et rapporte 4 euros.
- (a) Introduire une variable x_i à valeur dans $\{0, 1\}$ qui signifie prendre ou pas l'objet i , et modéliser le problème sous forme d'un problème de programmation linéaire en relâchant la contrainte $x_i \in \{0, 1\}$ en $0 \leq x_i \leq 1$.
- (b) Résoudre le problème de programmation linéaire.
- (c) Pour chaque solution non entière $x_i = n_i$, résoudre les deux problèmes obtenus en posant $x_i \leq n_i$ et $x_i \geq n_i$.
- Continuer jusqu'à trouver la solution du problème de départ.

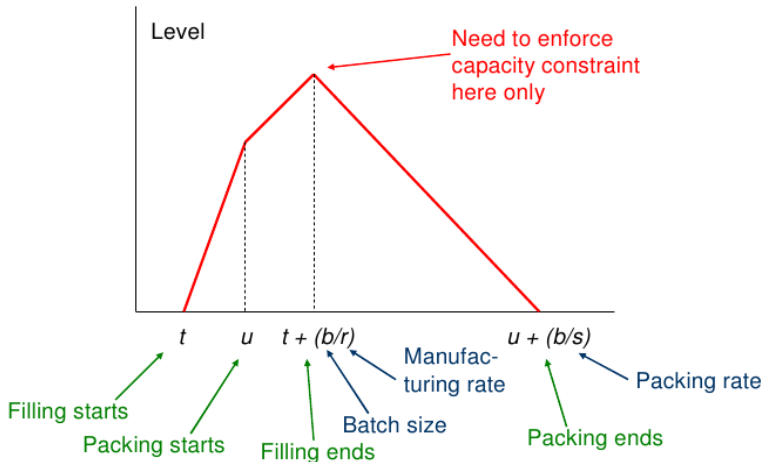
Stockage (Cumulative, 3)

- Planification de production avec stockage intermédiaire :



- Voir la page suivante.
- 👉 Donner le minimum de la date de l'achèvement !

Stockage (Cumulative, 3) (suite)



Stockage (Cumulative, 3) (suite)

- Une formulation avec des contraintes.

$$\min \quad T \quad \leftarrow (\text{makespan}) A \text{ minimiser}$$

$$s.t. \quad T \geq u_j + \frac{b_j}{s_j}, \forall j$$

$$t_j \geq R_j, \forall j \quad \leftarrow \text{temps fin tâche } t_j$$

$$\text{cumulative}(t, v, e, m) \quad \leftarrow m \text{ tankers de stockage}$$

$$v_i = u_i + \frac{b_i}{s_i} - t_i, \forall i \quad \leftarrow \text{duree des tâches } t_i$$

$$b_i \left(1 - \frac{s_i}{r_i}\right) + s_i u_i \leq C_i, \forall i \quad \leftarrow \text{capacite du tanker } i$$

$$\text{cumulative}(u, \left(\frac{b_1}{s_1}, \dots, \frac{b_n}{s_n}\right), e, p) \quad \leftarrow p \text{ units de paquetage}$$

$$u_j \geq t_j \geq 0 \quad e = (1, \dots, 1)$$

Contraintes Globales de Minizinc

(voir par famille plus loin)

- **Convention** en Minizinc

- $\$T$ ou $\$U$ représente tout type. $\$T$ peut être de type int, float, etc.
- Dans l'expression

function set of $\$U$: array_union(array [$\T] of set of $\$U$: x),

$\$U$ et $\$T$ peuvent être différents types mais dans la fonction

function set of $\$T$: 'intersect'(set of $\$T$: x, set of $\$T$: y)

tous les $\$$ sont du même type.

→ Différents noms de variables après $\$$ veulent dire qu'ils peuvent être de type différents. Alors que le même nom après $\$$ imposent d'avoir le même type.

- Exemple:

function set of float: array_union(array [int] of set of float: x)

Et

function set of int: 'intersect'(set of int: x, set of int: y).

- array [$\$T$] est en quelque sorte spécial et veut dire que le tableau est d'une dimension quelconque.

i.e array [int], array [int,int] ou array [int,int,int,int,int] etc.

Donc array [$\$T$] of set of $\$U$ veut dire qu'on peut avoir un tableau de dimension $\$T$, par exemple [int,int] (une matrice). Ce tableau est rempli d'ensembles de n'importe quel type. Par exemple, *sets of integers* comme dans 1,4,7,145.

⚠ *var int* et *int* sont différents. *int* est un paramètre entier tandis que *var int* est une variable de décision de type entier.

Contraintes Globales de Minizinc (suite)

La liste alphabétique des contraintes globales de MiniZinc.

- **alldifferent** (array[int] of var int: x)
alldifferent(array[int] of var set of int: x)
→ Constrains the array of objects x to be all different. Also available by the name all_different.
- alldifferent_except_0(array[int] of var int: x)
→ Constrains the elements of the array x to be all different except those elements that are assigned the value 0.
- all_disjoint(array[int] of var set of int: x)
→ Ensures that every pair of sets in the array x is disjoint.
- all_equal(array[int] of var int: x)
all_equal(array[int] of var set of int: x)
→ Constrains the array of objects x to have the same value.
- **among** (var int: n, array[int] of var int: x, set of int: v)
→ Requires exactly n variables in x to take one of the values in v.
- at_least(int: n, array[int] of var int: x, int: v)
at_least(int: n, array[int] of var set of int: x, set of int: v)
→ Requires at least n variables in x to take the value v. Also available by the name atleast.
- at_most(int: n, array[int] of var int: x, int: v)
at_most(int: n, array[int] of var set of int: x, set of int: v)
→ Requires at most n variables in x to take the value v. Also available by the name atmost.
- at_most1(array[int] of var set of int: s)
→ Requires that each pair of sets in s overlap in at most one element.
→ Also available by the name atmost1.
- **bin_packing** (int: c, array[int] of var int: bin, array[int] of int: w)
→ Requires that each item i be put into bin bin[i] such that the sum of the weights of each item, w[i], in each bin does not exceed the capacity c.
→ Aborts if an item has a negative weight or if the capacity is negative.
→ Aborts if the index sets of bin and w are not identical.

Contraintes Globales de Minizinc (suite)

- `bin_packing_capa`(array[int] of int: c, array[int] of var int: bin, array[int] of int:w)
→ Requires that each item i be put into bin $\text{bin}[i]$ such that the sum of the weights of each item, $w[i]$, in each bin b does not exceed the capacity $c[b]$. Aborts if an item has negative weight. Aborts if the index sets of bin and w are not identical.
- `bin_backing_load`(array[int] of var int: l, array[int] of var int: bin, array[int] of int: w)
→ Requires that each item i be put into bin $\text{bin}[i]$ such that the sum of the weights of each item, $w[i]$, in each bin b is equal to the load $l[b]$. Aborts if an item has negative weight. Aborts if the index sets of bin and w are not identical.
- `circuit` [array[int] of var int: x)
→ Constrains the elements of x to define a circuit where $x[i] = j$ mean that j is the successor of i .
- `count_eq` (array[int] of var int: x, var int: y, var int: c)
→ Constrains c to be the number of occurrences of y in x . Also available by the name `count`.
- `count_geq`(array[int] of var int: x, var int: y, var int: c)
→ Constrains c to greater than or equal to the number of occurrences of y in x .
- `count_gt`(array[int] of var int: x, var int: y, var int: c)
→ Constrains c to strictly greater than the number of occurrences of y in x .
- `count_leq`(array[int] of var int: x, var int: y, var int: c)
→ Constrains c to less than or equal to the number of occurrences of y in x .
- `count_lt`(array[int] of var int: x, var int: y, var int: c)
→ Constrains c to strictly less than the number of occurrences of y in x .
- `count_neq`(array[int] of var int: x, var int: y, var int: c)
→ Constrains c to not be the number of occurrences of y in x .
- `cumulative` (array[int] of var int: s, array[int] of var int: d, array[int] of var int: r, var int: b)
→ Requires that a set of tasks given by start times s , durations d , and resource requirements r , never require more than a global resource bound b at any one time. Aborts if s , d , and r do not have identical index sets. Aborts if a duration or resource requirement is negative.

Contraintes Globales de Minizinc (suite)

- decreasing(array[int] of var bool: x)
 decreasing(array[int] of var float: x)
 decreasing(array[int] of var int: x)
 decreasing(array[int] of var set of int: x)
 → Requires that the array x is in (non-strictly) decreasing order.
- diffn** (array[int] of var int: x, array[int] of var int: y, array[int] of var int: dx, array[int] of var int: dy)
 → Constrains rectangles, given by their origins x,y and sizes dx,dy, to be non-overlapping.
- disjoint**(var set of int: s, var set of int: t)
 → Requires that sets s and t do not intersect.
- distribute** (array[int] of var int: card, array[int] of var int: value, array[int] of var int: base)
 → Requires that card[i] is the number of occurrences of value[i] in base. In this implementation the values in value need not be distinct. Aborts if card and value do not have identical index sets.
- element** (var int: i, array[int] of var bool: x, var bool: y)
 element(var int: i, array[int] of var float: x, var float: y)
 element(var int: i, array[int] of var int: x, var int: y)
 element(var int: i, array[int] of var set of int: x, var set of int: y)
 → The same as $x[i] = y$. That is, y is the ith element of the array x.
- exactly** (int: n, array[int] of var int: x, int: v)
 exactly(int: n, array[int] of var set of int: x, set of int: v)
 → Requires exactly n variables in x to take the value v.
- global_cardinality** (array[int] of var int: x, array[int] of int: cover, array[int] of var int: counts)
 → Requires that the number of occurrences of cover[i] in x is counts[i]. Aborts if cover and counts do not have identical index sets.
- global_cardinality_closed**(array[int] of var int: x, array[int] of int: cover, array[int] of var int: counts)
 → Requires that the number of occurrences of cover[i] in x is counts[i]. The elements of x must take their values from cover. Aborts if cover and counts do not have identical index sets.
- global_cardinality_low_up**(array[int] of var int: x, array[int] of int: cover, array[int] of int: lb, array[int] of int: ub)
 → Requires that for all i, the value cover[i] appears at least lb[i] and at most ub[i] times in the array x.

Contraintes Globales de Minizinc (suite)

- `global_cardinality_low_up_closed(array[int] of var int: x, array[int] of int: cover, array[int] of int: lb, array[int] of int: ub)`
→ Requires that for all i , the value `cover[i]` appears at least `lb[i]` and at most `ub[i]` times in the array `x`. The elements of `x` must take their values from `cover`.
- `increasing(array[int] of var bool: x)`
`increasing(array[int] of var float: x)`
`increasing(array[int] of var int: x)`
`increasing(array[int] of var set of int: x)`
→ Requires that the array `x` is in (non-strictly) increasing order.

Channeling (assignment)

- `int_set_channel(array[int] of var int: x, array[int] of var set of int: y)`
→ Requires that $x[i] = j$ if and only if $i \in y[j]$.
- `inverse(array[int] of var int: f, array[int] of var int: invf)`
→ **Assign**
→
→ Constrains two arrays to represent inverse functions of each other. All the values in each array must be within the index set of the other array.
- `inverse_set(array[int] of var set of int: f, array[int] of var set of int: invf)`
→ Constrains the two arrays `f` and `invf` so that $j \in f[i]$ if and only if $i \in invf[j]$. All the values in each array's sets must be within the index set of the other array.
- `lex_greater(array[int] of var bool: x, array[int] of var bool: y)`
`lex_greater(array[int] of var float: x, array[int] of var float: y)`
`lex_greater(array[int] of var int: x, array[int] of var int: y)`
`lex_greater(array[int] of var set of int: x, array[int] of var set of int: y)`
→ Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.

Contraintes Globales de Minizinc (suite)

- `lex_greatereq(array[int] of var bool: x, array[int] of var bool: y)`
`lex_greatereq(array[int] of var float: x, array[int] of var float: y)`
`lex_greatereq(array[int] of var int: x, array[int] of var int: y)`
`lex_greatereq(array[int] of var set of int: x, array[int] of var set of int: y)`
 → Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.
- `lex_less(array[int] of var bool: x, array[int] of var bool: y)`
`lex_less(array[int] of var float: x, array[int] of var float: y)`
`lex_less(array[int] of var int: x, array[int] of var int: y)`
`lex_less(array[int] of var set of int: x, array[int] of var set of int: y)`
 → Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- `lex_lesseq(array[int] of var bool: x, array[int] of var bool: y)`
`lex_lesseq(array[int] of var float: x, array[int] of var float: y)`
`lex_lesseq(array[int] of var int: x, array[int] of var int: y)`
`lex_lesseq(array[int] of var set of int: x, array[int] of var set of int: y)`
 → Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- `lex2(array[int, int] of var int: x)`
 → Require adjacent rows and adjacent columns in the the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns may be equal.
- link_set_to_booleans** (var set of int: `s`, array[int] of var bool: `b`)
 → The array of booleans `b` is the characteristic representation of the set `s`. Aborts if the index set of `b` is not a superset of the possible values of `s`.
- maximum** (var int: `m`, array[int] of var int: `x`)
`maximum(var float: m, array[int] of var float: x)`
 → Constrains `m` to be the maximum of the values in `x`. (The array `x` must have at least one element.)

Contraintes Globales de Minizinc (suite)

- member** (array[int] of var bool: x, var bool: y)
 member(array[int] of var float: x, var float: y)
 member(array[int] of var int: x, var int: y)
 member(array[int] of var set of int: x, var set of int: y)
 member(var set of int: x, var int: y)
 → Requires that y occurs in the array or set x.
- minimum** (var float: m, array[int] of var float: x)
 minimum(var int: m, array[int] of var int: x)
 → Constrains m to be the minimum of the values in x. (The array x must have at least one element.)
- nvalue** (var int: n, array[int] of var int: x)
 → Requires that the number of distinct values in x is n.
- partition_set** (array[int] of var set of int: s, set of int: universe)
 → Partitions universe into disjoint sets.
- range** (array[int] of var int: x, var set of int: s, var set of int: t)
 → Requires that the image of function x (represented as an array) on set of values s is t. Aborts if $ub(s)$ is not a subset of the index set of x.
- regular** (array[int] of var int: x, int: Q, int: S, array[int,int] of int: d, int: q0, set of int: F)
 → The sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $\langle 1..Q, 1..S \rangle$ to $0..Q$) and initial state q0 (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state. Aborts if $Q < 1$. Aborts if $S < 1$. Aborts if the transition function d is not in $[1..Q, 1..s]$. Aborts if the start state, q0, is not in $1..Q$. Aborts if F is not a subset of $1..Q$.
- roots** (array[int] of var int: x, var set of int: s, var set of int: t)
 → Requires that $x[i] \in t$ for all $i \in s$. Aborts if $ub(s)$ is not a subset of the index set of x.
- sliding_sum** (int: low, int: up, int: seq, array[int] of var int: vs)
 → Requires that in each subsequence $vs[i], \dots, vs[i + seq - 1]$ the sum of the values belongs to the interval [low, up].
- sort** (array[int] of var int: x, array[int] of var int: y)
 → Requires that the multiset of values in x is the same as the multiset of values in y but y is in sorted order. Aborts if the cardinality of the index sets of x and y is not equal.

Contraintes Globales de Minizinc (suite)

- `strict_lex2(array[int, int] of var int: x)`
→ Require adjacent rows and adjacent columns in the the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns cannot be equal.
- `subcircuit (array[int] of var int: x)`
→ Constrains the elements of `x` to define a subcircuit where `x[i] = j` means that `j` is the successor of `i` and `x[i] = i` means that `i` is not in the circuit.
- `sum_pred (var int: i, array[int] of set of int: sets, array[int] of int: c, var int: s)`
→ Requires that the sum of `c[i1]...c[iN]` equals `s`, where `i1...iN` are the elements of the `i`th set in `sets`.
→ This constraint is usually named `sum`, but using that would conflict with the MiniZinc built-in function of the same name.
- `table (array[int] of var bool: x, array[int, int] of bool: t)`
`table(array[int] of var int: x, array[int, int] of int: t)`
→ Represents the constraint $x \in t$ where we consider each row in `t` to be a tuple and `t` as a set of tuples. Aborts if the second dimension of `t` does not equal the number of variables in `x`. The default decomposition of this constraint cannot be flattened if it occurs in a reified context.
- `value_precede (int: s, int: t, array[int] of var int: x)`
`value_precede(int: s, int: t, array[int] of var set of int: x)`
→ Requires that `s` precede `t` in the array `x`. For integer variables this constraint requires that if an element of `x` is equal to `t`, then another element of `x` with a lower index is equal to `s`. For set variables this constraint requires that if an element of `x` contains `t` but not `s`, then another element of `x` with lower index contains `s` but not `t`.
- `value_precede_chain(array[int] of int: c, array[int] of var int: x)`
`value_precede_chain(array[int] of int: c, array[int] of var set of int: x)`
→ Requires that the `value_precede` constraint is true for every pair of adjacent integers in `c` in the array `x`.

Contraintes globales par famille

La liste alphabétique des contraintes globales de MiniZinc.

* All-Different and related constraints

- predicate `all_different(array [int] of var int: x)`
Constrain the array of integers `x` to be all different.
- predicate `all_different(array [int] of var set of int: x)`
Constrain the array of sets of integers `x` to be all different.
- predicate `all_disjoint(array [int] of var set of int: S)`
Constrain the array of sets of integers `S` to be pairwise disjoint.
- predicate `all_equal(array [int] of var int: x)`
Constrain the array of integers `x` to be all equal
- predicate `all_equal(array [int] of var set of int: x)`
Constrain the array of sets of integers `x` to be all different
- predicate `alldifferent_except_0(array [int] of var int: vs)`
Constrain the array of integers `vs` to be all different except those elements that are assigned the value 0.
- function `var int: nvalue(array [int] of var int: x)`
Returns the number of distinct values in `x`.
- predicate `nvalue(var int: n, array [int] of var int: x)`
Requires that the number of distinct values in `x` is `n`.
- predicate `symmetric_all_different(array [int] of var int: x)`
Requires the array of integers `x` to be all different, and for all $i, x[i] = j \implies x[j] = i$.

Contraintes globales par famille (suite)

* Lexicographic constraints

- predicate `lex2(array [int,int] of var int: x)`
Require adjacent rows and adjacent columns in the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns may be equal.
- predicate `lex_greater(array [int] of var bool: x, array [int] of var bool: y)`
Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greater(array [int] of var int: x, array [int] of var int: y)`
Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greater(array [int] of var float: x, array [int] of var float: y)`
Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greater(array [int] of var set of int: x, array [int] of var set of int: y)`
Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greatereq(array [int] of var bool: x, array [int] of var bool: y)`
Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greatereq(array [int] of var int: x, array [int] of var int: y)`
Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greatereq(array [int] of var float: x, array [int] of var float: y)`
Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greatereq(array [int] of var set of int: x, array [int] of var set of int: y)`
Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.

Contraintes globales par famille (suite)

- predicate `lex_less(array [int] of var bool: x, array [int] of var bool: y)`
Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_less(array [int] of var int: x, array [int] of var int: y)`
Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_less(array [int] of var float: x, array [int] of var float: y)`
Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_less(array [int] of var set of int: x, array [int] of var set of int: y)`
Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_lesseq(array [int] of var bool: x, array [int] of var bool: y)`
Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_lesseq(array [int] of var float: x, array [int] of var float: y)`
Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_lesseq(array [int] of var int: x, array [int] of var int: y)`
Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_lesseq(array [int] of var set of int: x, array [int] of var set of int: y)`
Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `strict_lex2(array [int,int] of var int: x)`
Require adjacent rows and adjacent columns in the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns cannot be equal.

Contraintes globales par famille (suite)

En particulier les précédences de valeurs

- predicate `value_precede(int: s, int: t, array [int] of var int: x)`
Requires that `s` precede `t` in the array `x`.
Precedence means that if any element of `x` is equal to `t`, then another element of `x` with a lower index is equal to `s`.
- predicate `value_precede(int: s, int: t, array [int] of var set of int: x)`
Requires that `s` precede `t` in the array `x`.
Precedence means that if an element of `x` contains `t` but not `s`, then another element of `x` with lower index contains `s` but not `t`.
- predicate `value_precede_chain(array [int] of int: c, array [int] of var int: x)`
Requires that `c[i]` precedes `c[i + 1]` in the array `x`.
Precedence means that if any element of `x` is equal to `c[i + 1]`, then another element of `x` with a lower index is equal to `c[i]`.
- predicate `value_precede_chain(array [int] of int: c, array [int] of var set of int: x)`
Requires that `c[i]` precedes `c[i + 1]` in the array `x`.
Precedence means that if an element of `x` contains `c[i + 1]` but not `c[i]`, then another element of `x` with lower index contains `c[i]` but not `c[i + 1]`.

Contraintes globales par famille (suite)

* **Sorting constraints** **Et in/decreasing ponctuel**

- function `array [int] of var int: arg_sort(array [int] of var int: x)`
Returns the permutation p which causes x to be in sorted order hence $x[p[i]] \leq x[p[i + 1]]$. The permutation is the stable sort hence $x[p[i]] = x[p[i + 1]] \implies p[i] < p[i + 1]$.
- function `array [int] of var int: arg_sort(array [int] of var float: x)`
Returns the permutation p which causes x to be in sorted order hence $x[p[i]] \leq x[p[i + 1]]$. The permutation is the stable sort hence $x[p[i]] = x[p[i + 1]] \implies p[i] < p[i + 1]$.
- predicate `arg_sort(array [int] of var int: x, array [int] of var int: p)`
Constrains p to be the permutation which causes x to be in sorted order hence $x[p[i]] \leq x[p[i + 1]]$. The permutation is the stable sort hence $x[p[i]] = x[p[i + 1]] \implies p[i] < p[i + 1]$.
- predicate `arg_sort(array [int] of var float: x, array [int] of var int: p)`
Constrains p to be the permutation which causes x to be in sorted order hence $x[p[i]] \leq x[p[i + 1]]$. The permutation is the stable sort hence $x[p[i]] = x[p[i + 1]] \implies p[i] < p[i + 1]$.
- predicate `decreasing(array [int] of var bool: x)`
Requires that the array x is in decreasing order (duplicates are allowed).
- predicate `decreasing(array [int] of var float: x)`
Requires that the array x is in decreasing order (duplicates are allowed).
- predicate `decreasing(array [int] of var int: x)`
Requires that the array x is in decreasing order (duplicates are allowed).
- predicate `decreasing(array [int] of var set of int: x)`
Requires that the array x is in decreasing order (duplicates are allowed).
- predicate `increasing(array [int] of var bool: x)`
Requires that the array x is in increasing order (duplicates are allowed).
- predicate `increasing(array [int] of var float: x)`
Requires that the array x is in increasing order (duplicates are allowed).

Contraintes globales par famille (suite)

- predicate **increasing**(array [int] of var int: x)
Requires that the array x is in **increasing order** (duplicates are allowed).
- predicate **increasing**(array [int] of var set of int: x)
Requires that the array x is in **increasing order** (duplicates are allowed).
- function array [int] of var int: **sort**(array [int] of var int: x)
Return a multiset of values that is the same as the multiset of values in x but in sorted order.
- predicate **sort**(array [int] of var int: x, array [int] of var int: y)
Requires that the multiset of values in x are the same as the multiset of values in y but y is in sorted order.

Contraintes globales par famille (suite)

* Channeling constraints (assignment)

- predicate `int_set_channel`(array [int] of var int: x, array [int] of var set of int: y)
Requires that array of int variables x and array of set variables y are related such that $(x[i] = j) \iff (i \text{ in } y[j])$.
- function array [int] of var int: `inverse`(array [int] of var int: f)
Given a function f represented as an array, return the inverse function.
- predicate `inverse`(array [int] of var int: f, array [int] of var int: invf)
Constrains two arrays of int variables, f and invf, to represent inverse functions.
All the values in each array must be within the index set of the other array.
- predicate `inverse_set`(array [int] of var set of int: f, array [int] of var set of int: invf)
Constrains two arrays of set of int variables, f and invf, so that a j in f[i] iff i in invf[j].
All the values in each array's sets must be within the index set of the other array.
- predicate `link_set_to_booleans`(var set of int: s, array [int] of var bool: b)
Constrain the array of Booleans b to be a representation of the set s: $i \text{ in } s \iff b[i]$.
The index set of b must be a superset of the possible values of s.

Contraintes globales par famille (suite)

* Counting constraints

- function var int: **among**(array [int] of var int: x, set of int: v)
Returns the number of variables in x that take one of the values in v.
- predicate **among**(var int: n, array [int] of var int: x, set of int: v)
Requires exactly n variables in x to take one of the values in v.
- predicate **at_least**(int: n, array [int] of var int: x, int: v)
Requires at least n variables in x to take the value v.
- predicate **at_least**(int: n, array [int] of var set of int: x, set of int: v)
Requires at least n variables in x to take the value v.
- predicate **at_most**(int: n, array [int] of var int: x, int: v)
Requires at most n variables in x to take the value v.
- predicate **at_most**(int: n, array [int] of var set of int: x, set of int: v)
Requires at most n variables in x to take the value v.
- predicate **at_most1**(array [int] of var set of int: s)
Requires that each pair of sets in s overlap in at most one element.
- function var int: **count**(array [int] of var int: x, var int: y)
Returns the number of occurrences of y in x.
- predicate **count**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be the number of occurrences of y in x.
- predicate **count_eq**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be the number of occurrences of y in x.
- predicate **count_geq**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be greater than or equal to the number of occurrences of y in x.
- predicate **count_gt**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be strictly greater than the number of occurrences of y in x.
- predicate **count_leq**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be less than or equal to the number of occurrences of y in x.

Contraintes globales par famille (suite)

- predicate `count_lt`(array [int] of var int: x, var int: y, var int: c)
Constrains c to be strictly less than the number of occurrences of y in x.
- predicate `count_neq`(array [int] of var int: x, var int: y, var int: c)
Constrains c to be not equal to the number of occurrences of y in x.
- function array [int] of var int: `distribute`(array [int] of var int: value, array [int] of var int: base)
Returns an array of the number of occurrences of value[i] in base.
The values in value need not be distinct.
- predicate `distribute`(array [int] of var int: card, array [int] of var int: value, array [int] of var int: base)
Requires that card[i] is the number of occurrences of value[i] in base.
The values in value need not be distinct.
- predicate `exactly`(int: n, array [int] of var int: x, int: v)
Requires exactly n variables in x to take the value v.
- predicate `exactly`(int: n, array [int] of var set of int: x, set of int: v)
Requires exactly n variables in x to take the value v.
- function array [int] of var int: `global_cardinality`(array [int] of var int: x, array [int] of int: cover)
Returns the number of occurrences of cover[i] in x.
- predicate `global_cardinality`(array [int] of var int: x, array [int] of int: cover, array [int] of var int: counts)
Requires that the number of occurrences of cover[i] in x is counts[i].
- function array [int] of var int: `global_cardinality_closed`(array [int] of var int: x, array [int] of int: cover)
Returns an array with number of occurrences of i in x.
The elements of x must take their values from cover.
- predicate `global_cardinality_closed`(array [int] of var int: x, array [int] of int: cover, array [int] of var int: counts)
Requires that the number of occurrences of i in x is counts[i].
The elements of x must take their values from cover.

Contraintes globales par famille (suite)

- predicate `global_cardinality_low_up`(array [int] of var int: x, array [int] of int: cover,
array [int] of int: lbound, array [int] of int: ubound)
Requires that for all i, the value cover[i] appears at least lbound[i] and
at most ubound[i] times in the array x.
- predicate `global_cardinality_low_up_closed`(array [int] of var int: x, array [int] of int: cover,
array [int] of int: lbound, array [int] of int: ubound)
Requires that for all i, the value cover[i] appears at least lbound[i] and
at most ubound[i] times in the array x.
The elements of x must take their values from cover.

Contraintes globales par famille (suite)

* Packing constraints

- predicate `bin_packing`(int: c, array [int] of var int: bin, array [int] of int: w)
Requires that each item i with weight w[i], be put into bin[i] such that the sum of the weights of the items in each bin does not exceed the capacity c.

Assumptions: $\forall i, w[i] \geq 0, c \geq 0$

- predicate `bin_packing_capa`(array [int] of int: c, array [int] of var int: bin, array [int] of int: w)
Requires that each item i with weight w[i], be put into bin[i] such that the sum of the weights of the items in each bin b does not exceed the capacity c[b].

Assumptions: $\forall i, w[i] \geq 0, \forall b, c[b] \geq 0$

- function array [int] of var int: `bin_packing_load`(array [int] of var int: bin, array [int] of int: w)
Returns the load of each bin resulting from packing each item i with weight w[i] into bin[i], where the load is defined as the sum of the weights of the items in each bin.

Assumptions: $\forall i, w[i] \geq 0$

- predicate `bin_packing_load`(array [int] of var int: load, array [int] of var int: bin, array [int] of int: w)
Requires that each item i with weight w[i], be put into bin[i] such that the sum of the weights of the items in each bin b is equal to load[b].

Assumptions: $\forall i, w[i] \geq 0$

- predicate `diffn`(array [int] of var int: x, array [int] of var int: y, array [int] of var int: dx, array [int] of var int: dy)
Constrains rectangles i, given by their origins (x[i], y[i]) and sizes (dx[i], dy[i]), to be non-overlapping.
Zero-width rectangles can still not overlap with any other rectangle.

- predicate `diffn_k`(array [int,int] of var int: box_posn, array [int,int] of var int: box_size)
Constrains k-dimensional boxes to be non-overlapping.
For each box i and dimension j, box_posn[i, j] is the base position of the box in dimension j, and box_size[i, j] is the size in that dimension.
Boxes whose size is 0 in any dimension still cannot overlap with any other box.

Contraintes globales par famille (suite)

- predicate `diffn_nonstrict`(array [int] of var int: x, array [int] of var int: y, array [int] of var int: dx, array [int] of var int: dy)
 Constrains rectangles i, given by their origins (x[i], y[i]) and sizes (dx[i], dy[i]), to be non-overlapping.
 Zero-width rectangles can be packed anywhere.
- predicate `diffn_nonstrict_k`(array [int,int] of var int: box_posn, array [int,int] of var int: box_size)
 Constrains k-dimensional boxes to be non-overlapping.
 For each box i and dimension j, box_posn[i, j] is the base position of the box in dimension j, and box_size[i, j] is the size in that dimension.
 Boxes whose size is 0 in at least one dimension can be packed anywhere.
- predicate `geost`(int: k, array [int,int] of int: rect_size, array [int,int] of int: rect_offset, array [int] of set of int: shape, array [int,int] of var int: x, array [int] of var int: kind)
 A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap.

Parameters

- k: the number of dimensions
- rect_size: the size of each box in k dimensions
- rect_offset: the offset of each box from the base position in k dimensions
- shape: the set of rectangles defining the i-th shape.
 → Assumption: Each pair of boxes in a shape must not overlap.
- x: the base position of each object. x[i,j] is the position of object i in dimension j.
- kind: the shape used by each object.

Contraintes globales par famille (suite)

- predicate `geost_bb`(int: k, array [int,int] of int: rect_size, array [int,int] of int: rect_offset, array [int] of set of int: shape, array [int,int] of var int: x, array [int] of var int: kind, array [int] of var int: l, array [int] of var int: u)

A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap, and that all objects fit within a global k dimensional bounding box.

Parameters

- k: the number of dimensions
- rect_size: the size of each box in k dimensions
- rect_offset: the offset of each box from the base position in k dimensions
- shape: the set of rectangles defining the i-th shape.
→ Assumption: Each pair of boxes in a shape must not overlap.
- x: the base position of each object. $x[i,j]$ is the position of object i in dimension j.
- kind: the shape used by each object.
- l: is an array of lower bounds, $l[i]$ is the minimum bounding box for all objects in dimension i.
- u: is an array of upper bounds, $u[i]$ is the maximum bounding box for all objects in dimension i.
- predicate `geost_smallest_bb`(int: k, array [int,int] of int: rect_size, array [int,int] of int: rect_offset, array [int] of set of int: shape, array [int,int] of var int: x, array [int] of var int: kind, array [int] of var int: l, array [int] of var int: u)

A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap, and that all objects fit within a global k dimensional bounding box. In addition, it enforces that the bounding box is the smallest one containing all objects, i.e., each of the $2k$ boundaries is touched by at least by one object.

Parameters

- k: the number of dimensions
- rect_size: the size of each box in k dimensions
- rect_offset: the offset of each box from the base position in k dimensions
- shape: the set of rectangles defining the i-th shape.
→ Assumption: Each pair of boxes in a shape must not overlap.
- x: the base position of each object. $x[i,j]$ is the position of object i in dimension j.
- kind: the shape used by each object.
- l: is an array of lower bounds, $l[i]$ is the minimum bounding box for all objects in dimension i.
- u: is an array of upper bounds, $u[i]$ is the maximum bounding box for all objects in dimension i.

Contraintes globales par famille (suite)

- predicate `knapsack(array [int] of int: w, array [int] of int: p, array [int] of var int: x, var int: W, var int: P)`
Requires that items are packed in a knapsack with certain weight and profit restrictions.

Assumptions:

- Weights w and profits p must be non-negative
- w , p and x must have the same index sets

Parameters

- w : weight of each type of item
- p : profit of each type of item
- x : number of items of each type that are packed
- W : sum of sizes of all items in the knapsack
- P : sum of profits of all items in the knapsack

Contraintes globales par famille (suite)

* Scheduling constraints

- predicate **alternative**(var opt int: s0, var int: d0, array [int] of var opt int: s, array [int] of var int: d)
Alternative constraint for optional tasks.
Task (s0,d0) spans the optional tasks (s[i],d[i]) in the array arguments and at most one can occur
- predicate **cumulative**(array [int] of var opt int: s, array [int] of var int: d, array [int] of var int: r, var int: b)
Requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.
Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions: $\forall i, d[i] \geq 0 \wedge r[i] \geq 0$

- predicate **cumulative**(array [int] of var int: s, array [int] of var int: d, array [int] of var int: r, var int: b)
Requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.

Assumptions: $\forall i, d[i] \geq \wedge r[i] \geq 0$

- predicate **disjunctive**(array [int] of var opt int: s, array [int] of var int: d)
Requires that a set of tasks given by start times s and durations d do not overlap in time.
Tasks with duration 0 can be scheduled at any time, even in the middle of other tasks.
Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions: $\forall i, d[i] \geq 0$

- predicate **disjunctive**(array [int] of var int: s, array [int] of var int: d)
Requires that a set of tasks given by start times s and durations d do not overlap in time.
Tasks with duration 0 can be scheduled at any time, even in the middle of other tasks.

Assumptions: $\forall i, d[i] \geq 0$

Contraintes globales par famille (suite)

- predicate `disjunctive_strict`(array [int] of var opt int: s, array [int] of var int: d)
Requires that a set of tasks given by start times s and durations d do not overlap in time.
Tasks with duration 0 CANNOT be scheduled at any time, but only when no other task is running.
Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions: $\forall i, d[i] \geq 0$

- predicate `disjunctive_strict`(array [int] of var int: s, array [int] of var int: d)
Requires that a set of tasks given by start times s and durations d do not overlap in time.
Tasks with duration 0 CANNOT be scheduled at any time, but only when no other task is running.

Assumptions: $\forall i, d[i] \geq 0$

- predicate `span`(var opt int: s0, var int: d0, array [int] of var opt int: s, array [int] of var int: d)
Span constraint for optional tasks. Task (s0,d0) spans the **optional tasks** (s[i],d[i]) in the array arguments.

Contraintes globales par famille (suite)

* Extensional constraints (table, regular etc.)

- predicate **regular**(array [int] of var int: x, int: Q, int: S, array [int,int] of int: d, int: q0, set of int: F)
The sequence of values in array x (which must all be in the range 1..S) is accepted by the DFA of Q states with input 1..S and transition function d (which maps (1..Q, 1..S) \rightarrow 0..Q)) and initial state q0 (which must be in 1..Q) and accepting states F (which all must be in 1..Q).
We reserve state 0 to be an always failing state.
- predicate **regular_nfa**(array [int] of var int: x, int: Q, int: S, array [int,int] of set of int: d, int: q0, set of int: F)
The sequence of values in array x (which must all be in the range 1..S) is accepted by the NFA of Q states with input 1..S and transition function d (which maps (1..Q, 1..S) \rightarrow set of 1..Q)) and initial state q0 (which must be in 1..Q) and accepting states F (which all must be in 1..Q).
- predicate **table**(array [int] of var bool: x, array [int,int] of bool: t)
Represents the constraint x in t where we consider each row in t to be a tuple and t as a set of tuples.
- predicate **table**(array [int] of var int: x, array [int,int] of int: t)
Represents the constraint x in t where we consider each row in t to be a tuple and t as a set of tuples.

Installer GLPK

- Vous pourriez être intéressé par un autre outil de Programmation Mathématique : GLPK.

- **Installation sur Windows** : site

<http://sourceforge.net/projects/winglpk/files/winglpk/GLPK-4.47.1/>

- En décembre 2012, *winglpk-4.47.1*

- Interface graphique (GUI) pour glpk version Windows :

→ Récupérer l'outil "Gusek" sur le site

<http://gusek.sourceforge.net/gusek.html>

→ Décompresser, Installer, lancer Gusek

→ Une fenêtre s'ouvre (éditeur, résultats dans le cadre droit)

→ Dans les options, préciser que vous utilisez GMPL (.mod)

→ ouvrir ou créer un fichier (par ex., **pb.mod**)

→ L'icône "go" pour lancer glpsol sur le problème actuel

→ Pour le fichier de données, *"Tools > Use External .dat"*

→ A droite, vous pouvez taper les commandes :

glpsol -model pb.mod [-data pb.dat -output pb.out] (entre []=optionnel)

- Il y a aussi "GLPK Lab" pour Windows.

Installer GLPK (suite)

- **Installation sur Linux** : site

<http://ftp.gnu.org/gnu/glpk/glpk-4.47.tar.gz>

- Décompresser, installer
- Editer, créer le fichier **pb.mode**
- Dans une fenêtre "terminal", taper (entre [] = si c'est le cas)
`glpsol -model pb.mod [-data pb.dat -output pb.out]`

- **Installation sur Mac** :

- Si vous savez compiler sur Mac, suivre les instructions sur le site
<http://www.arnab-deka.com/posts/2010/02/installing-glpk-on-a-mac/>
- Si vous connaissez *darwinport*, installer glpk dernière version avec la commande "port" (ou l'outil *portauthority*)
- Si vous voulez juste l'exécutable (suffit pour ce BE), une solution "maline" est de récupérer Coliop3 qui contient glpk.
- ☞ Cette méthode fonctionne également pour les plateformes Linux et Windows. Son inconvénient est qu'on ne pourra pas utiliser *glpk* depuis un programme C/C++.

Installer GLPK (suite)

- **Passage par Coliop3 (toute plateformes) :**
 - Récupérer Coliop3 depuis le site
<http://www.coliop.org/download/Cmpl-1.7.1-osx.zip>
 - Décompresser et récupérer *glpsol* (l'exécutable pour ce BE) dans le répertoire Cmpl/Thirdparty/GLPK/glpsol
 - Vous pouvez ensuite récupérer les exemples et la documentation depuis n'importe quelle distribution de Glpk (par exemple Linux).
 - Utilisation :
 - Exécuter l'application "Terminal" (dans /Applications/Utilities/)
 - Placez vous dans le répertoire Cmpl/Thirdparty/GLPK
 - Avec un éditeur (*textedit*), créer ou éditer le fichier **pb.mod**
 - Dans la fenêtre "terminal", taper
`./glpsol -model pb.mod [-data pb.dat -output pb.out]` (entre [] = optionnel)
- Quelques documents déposés avec le sujet de ce BE.
- Pour Windows, il y a aussi "GLPK Lab".

Alternative : Installation de LPSOLVE

Si vous le souhaitez, pour récupérer LPSOLVE, aller sur la page :

<http://sourceforge.net/projects/lpsolve/files/lpsolve/5.5.2.0>

- Windows : récupérer le fichier exécutable `lp_solve_5.5.2.0_IDE_Setup.exe`
 - Linux (32 bits) : `lp_solve_5.5.2.0_exe_ux32.tar.gz` (et `...u64...` pour 64 bits)
 - Mac : `lp_solve_5.5.2.0_exe_osx32.tar.gz`
 - Doc (il y a la doc disponible sur le Web) : `lp_solve_5.5.2.0_doc.tar.gz`
 - Pour s'en servir sous Excel : Voir la même page.
 - Pour OpenOffice : une extension existe chez OpenOffice.
 - ➔ Voir dans les fichiers déposés sur Pédagogie.
 - Il est possible d'invoquer LPSOLVE depuis un programme :
 - ➔ récupérer sur la page ci-dessus le fichier `lp_solve_5.5.2.0_c.tar.gz`
- 📖 Lire le fichier (court) **lp-sove-format.txt** qui vous permet une prise en main rapide.

Réels : Primal-Dual

Rappel du cours : pour un programme linéaire général :

$$\begin{array}{ll} \text{Min } c^T x & \text{sur le vecteur } x \geq 0 \\ \text{s.t. } Ax \geq b. \end{array}$$

Le dual sera :

$$\begin{array}{ll} \text{Max } b^T \alpha & \text{sur le vecteur } \alpha \\ \text{s.t. } \alpha_i \geq 0 \text{ et } A^T \alpha \leq c. \end{array}$$

Symétriquement :

➔ Si dans Primal, $\text{Max}\{C^T x | Ax \leq b, x \geq 0\}$

➔ Dans Dual : $\text{Min}\{b^T \alpha | A^T \alpha \geq c, \alpha \geq 0\}$

☞ α représente le vecteur des variables de la forme Duale (on utilise souvent y comme dans les exemples).

☞ Pour les intérêts de ces deux formes, voir cours.

Réels : Primal-Dual (suite)

Exemple : pour la forme primale

Fonction Objective : **maximize** $2*x[1]+3*x[2]$;
Les contraintes :
 $4*x[1]+8*x[2] \leq 12$;
 $2*x[1]+x[2] \leq 3$;
 $3*x[1]+2*x[2] \leq 4$;

On aura (à maximiser):

$$A = \begin{bmatrix} 4 & 8 \\ 2 & 1 \\ 3 & 2 \end{bmatrix} x \leq b = \begin{bmatrix} 12 \\ 3 \\ 4 \end{bmatrix} \text{ avec } c^T = \begin{bmatrix} 2 \\ 3 \end{bmatrix} x \text{ (maximiser)}$$

Ce qui donnera la forme Duale (à minimiser):

$$A^T = \begin{bmatrix} 4 & 2 & 3 \\ 8 & 1 & 2 \end{bmatrix} y \geq c = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \text{ avec } b^T = [12 \quad 3 \quad 4] y \text{ (minimiser)}$$

C'est à dire :

Fonction Objective : **minimize** $12*y[1]+3*y[2]+4*y[3]$;
Les contraintes :
 $4*y[1]+2*y[2]+3*y[3] \geq 2$;
 $8*y[1]+y[2]+2*y[3] \geq 3$;

Réels : Primal-Dual (suite)

- Créer le problème (forme Primale) suivant :
 - ➔ Remarquer l'utilisation d'un vecteur simple.
- ☞ Préciser les réels constantes sinon on aura des messages d'erreur de type.

```

set of 1..2 : s = 1.. 2;
array[s] of var float : x;
var float : obj;

constraint
  forall(i in s) (x[i] >= 0.0);

constraint
  4.0*x[1]+8.0*x[2] <= 12.0 /\
  2.0*x[1]+x[2] <= 3.0 /\
  3.0*x[1]+2.0*x[2] <=4.0;

constraint
  obj = 2.0*x[1]+3.0*x[2];

solve maximize obj;

output ["x=", show(x), " objective=" , show(obj)];

%mnz-g12mip primal.mzn (ou minizinc -b mip primal.mzn)
%x=[0.49999999999999983, 1.25] objective=4.75

```

Réels : Primal-Dual (suite)

- La forme duale du même problème :

```

set of int : J = 1.. 3;      % ou set of 1..3 : J = 1.. 3;
array[J] of var float : y;
var float : obj;

solve minimize obj;

constraint
  forall(i in J) (y[i] >= 0.0);
constraint
  obj = 12.0*y[1]+3.0*y[2]+4.0*y[3];

constraint
  4.0*y[1]+2.0*y[2]+3.0*y[3] >= 2.0
  ^
  8.0*y[1]+y[2]+2.0*y[3] >=3.0;

output ["y=", show(y), " objective=" , show(obj)];

%mnz-g12mip dual.mzn
%y=[0.31245, 0.0, 0.25] objective=4.75

```

☞ On remarque que les *extrema* sont identiques !

Modélisation Giapetto

- L'atelier de *Giapetto* fabrique deux 2 de jouets en bois : **soldats** et **trains**.
- Un soldat se vend à 27 euros et utilise 10 euros de matières premières.
Chaque soldat coûte par ailleurs 14 euros en coûts divers (salaire, amortissement, etc.).
- Un train se vend 21 euros et utilise 9 euros en matières premières.
Chaque train coûte en frais généraux 10 euros.
- La fabrication des soldats et des trains en bois exige deux types de machines pour la menuiserie et la finition.
- Un soldat a besoin de 2 heures de finition et de 1 heure de menuiserie.
Un train a besoin de 1 heure de finition et 1 heure de menuiserie.
- L'entreprise dispose de toute la matière première. Par contre, elle ne dispose que de 100 heures de finition et 80 heures de menuiserie.
- La demande des trains est illimitée, mais au plus 40 soldats sont achetés chaque semaine.
- *Giapetto* veut maximiser ses bénéfices d'hebdomadaire.

Modélisation Giapetto (suite)

- Soit x_1 : nbr de soldats en bois produits par semaine et
 x_2 : nbr de trains produits par semaine.

- Les bénéfices seront alors

$$Z = (27 - 10 - 14)x_1 + (21 - 9 - 10)x_2 = 3x_1 + 2x_2$$

- Par ailleurs, les contraintes de production sont :

$$\begin{array}{ll} \infty \geq x_1 \geq 0 & \text{et} \quad 40 \geq x_2 \geq 0 \text{ (nbr de produits fabriqués)} \\ x_1 + x_2 \leq 80 \text{ (menuiseries)} & \text{et} \quad 2x_1 + x_2 \leq 100 \text{ (finition)} \end{array}$$

Un premier test fait en CLP :

```
Z=3*X1+2*X2 ,
X1::0..40, X2 >= 0,
X1+X2 =< 80, 2*X1+X2 =< 100,
maxof(labeling([X1,X2]),Z).

Test :
Z = 180
X1 = 20
X2 = 60
```

- Solution Minizinc

... ~>

Modélisation Giapetto (suite)

Une solution Minizinc (*giapetto.mzn*)

```
% problème giapetto
% Decision variables
var int : x1; % soldat
var int : x2 ; % train

% Objective function
solve maximize 3*x1 + 2*x2;
% Constraints
constraint
  x1 >= 0 /\ x2 >= 0
  /\ 2*x1 + x2 <= 100 % Finition
  /\ x1 + x2 <= 80 % Menuiserie
  /\ x1 <= 40; % Demande

output ["x1 = " , show(x1), " , x2 = " , show(x2), " , objective = "
  , show(3*x1 + 2*x2), "\n"];

%Exécuter minizinc giapetto.mzn
% x1 = 20, x2 = 60, objective = 180
```

Modèle et Data pour Giapetto

- Minizinc peut fonctionner sur une base plus souple de Modèle + Data.
- Le Modèle peut être figé mais le Data peut varier,
- La partie données du problème peut être insérée à la suite du modèle ou être de préférence placée dans un autre fichier.
- Pour le même problème *Giapetto*, on a le modèle Minizinc suivi de ses paramètres (dans le même fichier, comme dans l'exemple de coloration) :

```

set of int : TOY; % Pour voir les valeurs, regarder ci-dessous !

% Parameters
array[TOY] of int : Heures_Finition;
array[TOY] of int : Heures_Menuiserie;
array[TOY] of int : Demande_toys;
array[TOY] of int : Profit_toys;

% Decision variables
array[TOY] of var int : x;
var int : obj = sum(i in TOY) (Profit_toys[i]*x[i]);
%var x (i in TOY) >=0;

% Objective function
solve maximize obj;

```

Modèle et Data pour Giapetto (suite)

```

constraint
  sum(i in TOY) (Heures_Finition[i]*x[i]) <= 100; % Fin_heures

constraint
  sum(i in TOY) (Heures_Menuiserie[i]*x[i]) <= 80; % Menus_heures

constraint
  forall(i in TOY) (x[i] <= Demande_toys[i]);

output ["Obj=", show(obj), "\n"];

%-----
% la partie Data

TOY = 1..2;
Heures_Finition=[1,2];
Heures_Menuiserie=[1,1];
Demande_toys = [40, 6000000];
Profit_toys = [3,2];

%-----
% Un test : Obj=180

```

Modèle et Data pour Giapetto (suite)

- On peut donc placer la partie Data dans un autre fichier (Giapetto.dzn) :

```
% la partie Data

TOY = 1..2;

Heures_Finition=[1,2];

Heures_Menuiserie=[1,1];

Demande_toys = [40, 6000000];

Profit_toys = [3,2];

%-----
% Un test : Obj=180
```

- Si on souhaite tester le modèle sur d'autres données, on change seulement de fichier Data; le modèle ne change pas.
- Exemple tiré du document de prise en main de Glpk (Gnu Linear Programming Kit) "*Introduction to linear optimization*", sur le site de GLPK.

Coloration plus générale

- On peut généraliser le modèle de coloration sachant que le graphe des incompatibilités est donné sous forme d'un tableau (matrice).
 - Une matrice $M : \text{noeuds} \times \text{couleur}$ contiendra le résultat final où $M[n, c]$ contiendra 1 si le noeud n reçoit la couleur c , 0 sinon.
- Avant de regarder le code, réfléchir au modèle !

```

int: nb_noeuds=5;           % nombre de noeuds

set of int: noeuds = 1..nb_noeuds; % ensemble de noeuds
int: nb_aretes=5;           % le graphe a 5 arêtes

% Le graphe :
array[1..nb_aretes, 1..2] of noeuds: aretes = [|      % remarquer les '|'
1, 5|
2, 3|
2, 4|
3, 5|
4, 5|
|];

% On sait que 4 couleurs suffisent largement
int: nb_couleurs = 4;

% matrice_noeud_couleur[i,c] = 1 veut dire : le noeud i a reçu la couleur c
array[noeuds, 1..nb_couleurs] of var 0..1: matrice_noeud_couleur;

solve satisfy;

```

Coloration plus générale (suite)

```

constraint
% CONTROLE : pas de loop dans le graphe (d'un noeud à lui même)
forall(i in 1..nb_aretes) (
    aretes[i,1] != aretes[i,2]
)
^
% Chaque noeud reçoit une seule couleur :
%      pour toute ligne de la matrice, la somme =1
forall(i in noeuds)
    (sum(c in 1..nb_couleurs) (matrice_noeud_couleur[i,c]) = 1)
^
% Deux adjacents n'auront pas la même couleur : Dans une colonne de
% la matrice, la somme des valeurs de deux noeuds voisins <= 1
forall(i in 1..nb_aretes, c in 1..nb_couleurs) (
    matrice_noeud_couleur[aretes[i,1],c] + matrice_noeud_couleur[aretes[i,2],c] <= 1
);

output ["obj: ", "\n",]
++ [" c1"++" c2"++" c3"++" c4"]
++ [if j = 1 then "\n" ++ show(i) ++ ": " else " " endif ++
    show(matrice_noeud_couleur[i,j])++" " | i in noeuds, j in 1..nb_couleurs] ++ ["\n"];
%-----
/* Solution
   c1 c2 c3 c4
1: 0 1 0 0      ← la couleur 2 donnée aux noeud 1, 3 et 4
2: 1 0 0 0      ← la couleur 1 donnée aux noeud 2 et 5
3: 0 1 0 0
4: 0 1 0 0
5: 1 0 0 0
*/

```

- Explications sur le tableau, *output*, *set*,

Coloration plus générale (suite)

Remarque sur la toute dernière contrainte sur la `matrice_noeud_couleur` (appelons la **M**) :

```

ne pas écrire  M[aretes[i,1],c] != M[aretes[i,2],c]
qui on impose que l'un des deux =1,
Or, ceci n'est pas toujours le cas (cette couleur est peut être non utilisée)
Par contre, on peut dire :
Si cette couleur est utilisée (il y a un "1" sur une des lignes de cette colonne)
Alors il faut 2 valeurs différentes pour les 2 noeuds de l'arête :
    (sum(j in 1..nb_noeuds) (matrice_noeud_couleur[j,c])) > 0
        -> % implication
        (M[aretes[i,1],c] != M[aretes[i,2],c])

```

- Remarques sur l'implication.

Optimisation du nombre de couleurs

On reprend le code précédent pour proposer une solution qui minimise le nombre de couleurs utilisées.

- L'idée : compter le nombre de fois où une couleur est utilisée (pour au moins un noeud).
- Ajouter au code précédent la partie ci-dessous et compléter : on compte le nombre de colonnes de la matrice où au moins un "1" apparaît.

```
solve minimize nb_coul_used; % replace solve satisfy;
%....
var int : nb_coul_used = sum(c in 1..nb_couleurs) (
    bool2int(sum(v in noeuds) (matrice_noeud_couleur[v,c]) > 0)
);

% Et ajouter l'affichage de la variable nb_coul_used
```

- Dans le "morceau" de code ci-dessus, on compte, dans la matrice *noeuds* × *couleurs* le nombre de colonnes où au moins un "1" apparaît.

→ Le test `sum(v in noeuds) (matrice_noeud_couleur[v,c]) > 0` donne un résultat booléen et il faut donc le convertir en un entier (vrai=1, faux=0) via `bool2int` pour pouvoir faire une somme sur ces valeurs.

Optimisation du nombre de couleurs (suite)

Une 2e manière d'optimiser le nombre de couleurs :

- Reprendre le code précédent (sans l'optimisation ci-dessus) pour proposer une autre solution qui minimise le nombre de couleurs utilisées.
- Ajouter à ce code la partie ci-dessous et compléter : on compte le nombre de colonnes de la matrice où au moins un "1" apparaît et on stock le résultat dans un vecteur.
→ Remplacer également "solve satisfy" par le "solve" donné ci-dessus.
- Remarques sur l'implication (">") et la méthode de calcul, minimisation.

```
var int : nb_coul_used = sum(c in 1..nb_couleurs) (vect_used_colors[c]);

solve minimize nb_coul_used;

array[1..nb_couleurs] of var int : vect_used_colors;

constraint
% Si une colonne c porte au moins un "1", mettre 1 dans vect_used_colors[c]
forall(c in 1..nb_couleurs)
(
  (((sum(i in 1..nb_noeuds) (matrice_noeud_couleur[i,c])) > 0) -> (vect_used_colors[c]
]=1))
  % Important de mettre 0
  ^ (((sum(i in 1..nb_noeuds) (matrice_noeud_couleur[i,c])) = 0) -> (vect_used_colors[c]
]=0))
);
```

Séparation Modèle-Data

- On reprend l'exemple de coloration vu plus haut.
 - Cette fois, on sépare le modèle des données (le modèle est ci-dessous)
 - Dans cette version, les données sont détachées du modèle mais placées dans le même fichier.
- Il suffira ensuite de placer la partie Data dans un autre fichier pour résoudre une autre instance de ce problème.

```

int: n;           % nombre de noeuds

set of int: V = 1..n; % set of noeuds

int: num_edges;

array[1..num_edges, 1..2] of V: E;

% 4 couleurs suffisent largement
int: nc = 4;

% x[i,c] = 1 veut dire : le noeud i prend la couleur c
array[V, 1..nc] of var 0..1: x;

solve satisfy;

constraint

```

Séparation Modèle-Data (suite)

```
% CONTROLE : pas de loop dans les donnée
forall(i in 1..num_edges) (
  E[i,1] != E[i,2]
)

^
% Tout noeud reçoit une couleur
forall(i in V) (sum(c in 1..nc) (x[i,c]) = 1)
^
% Les noeuds adjacents ne peuvent pas avoir la même couleur
forall(i in 1..num_edges, c in 1..nc) (
  x[E[i,1],c] + x[E[i,2],c] <= 1
)
;

output ["obj: ", "\n",]
++[" if j = 1 then "\n" ++ show(i) ++ ": " else " " endif ++
  show(x[i,j]) | i in V, j in 1..nc]
++ ["\n"];
%-----
```

Et les données (placées dans un fichier de données qui sera donné en paramètre du solveur ou placées après le modèle) :

Séparation Modèle-Data (suite)

```
% data  
  
% La solution optimale : 4  
  
n = 5;  
num_edges = 5;  
  
E = array2d(1..num_edges, 1..2, [  
    1, 5,  
    2, 3,  
    2, 4,  
    3, 5,  
    4, 5  
]);
```

- Il suffit donc de découper la partie Data, la placer dans (par exemple) le fichier *graphe.dzn* puis de lancer *minizinc coloration.mzn graphe1.dzn*.
→ Puis plus tard avec *graphe2.dzn*,

PERT (Big M)

- La méthode Big-M a été abordée en cours.

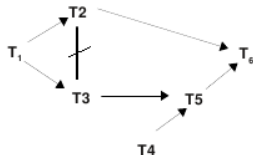
- On veut les dates de début des tâches.

- Propagations des contraintes de l'exemple :

Domaines : $T_1, T_2, T_3, T_4, T_5, T_6 \in 1..5$

Contraintes : $T_1 < T_2, T_1 < T_3, T_2 \neq T_3,$
 $T_2 < T_6, T_3 < T_5, T_4 < T_5, T_5 < T_6.$

- ☞ Minimiser T_i (en l'occurrence T_6)



$$T_1 < T_2$$

$$T_1 < T_3$$

$$T_2 \uparrow T_3$$

$$T_2 < T_6$$

$$T_3 < T_5$$

$$T_4 < T_5$$

$$T_5 < T_6$$

Remarques sur la Programmation Linéaire (LPSolve) :

- Traite $x < y$ comme $x \leq y$: utiliser $x < y - \varepsilon$ (selon les domaines)
- Comment dire $X \neq Y$: résoudre les 2 inégalités $X > Y$ or $Y > X$!
- Créer le fichier pert.mzn avec le contenu suivant.

- ☞ La méthode **BigM** utilisée pour les disjonctions.

PERT (Big M) (suite)

```
% Pert

par int : nbtaches = 6; % me mot clef "par" = parameter (optionnel ici)
set of int : j = 1..nbtaches;
array[j] of var int : debutsAuPlusTot; % >=0, <=50, default 0; les débuts
array[j] of var int : debttaches; % >=0;
array[j] of var int : fintaches; % >=0 ;

var int : z;

var 0..1 : y; % Binaire (au lieu de bool, utilisée pour BigM)

int : BigM = 6; % 6 = somme des durées + max(fintaches)

constraint
forall(i in j) (
    debutsAuPlusTot[i] >= 0
    ^ debutsAuPlusTot[i] <= 50
    ^ debttaches[i] >= 0
    ^ fintaches[i] >= 0
);

constraint
forall(i in j) (fintaches[i] >= debttaches[i]+1);

constraint
fintaches[1] <= debttaches[2] % T1 < T2
^ fintaches[1] <= debttaches[3] % T1 < T3
^ fintaches[2] <= debttaches[6] % T2 < T6
^ fintaches[3] <= debttaches[5] % T3 < T5
^ fintaches[4] <= debttaches[5] % T4 < T5
^ fintaches[5] <= debttaches[6] % T5 < T6
;
```

PERT (Big M) (suite)

```
% Utiliser BIG-M pour exprimer une différence (!= non lineaire en LP)
% A != B : B<=A-1+BigM*Y, A <= B-1+BigM*(1-Y)
% Préférer "<" car la partie droite augmente et il faut rester "<=".
```

% On peut faire simplement debtaches[2] != debtaches[3];
% Pour utiliser BigM : on traduit la contrainte ci-dessus en
% debtaches[2] > debtaches[3] OR debtaches[3] > debtaches[2]
% puis on passe de "<" à "<=" non autorisé dans LP puis on utilise la technique BIGM :

```
constraint
    debtaches[2] <= debtaches[3] + BigM*(1-y)-1
    /\ debtaches[3] <= debtaches[2] + BigM*y -1;

constraint forall(i in j) (z >= fintaches[i]);

solve minimize z;
```

% utiliser "++" (au lieu de ',') pour écrire si l'écriture utilise des indices

```
output [ "tache debut fin\n"
    ++ [ show(i) ++ " " ++ show(debtaches[i]) ++ " " ++ show(fintaches[i]) ++ "\n" |
        i in j ]
    ++ [ "\n" ];
```

/ une sol trouvée (pour les deux : BigM ou '!=') :*

```
tache debut fin
1 - 0 - 1
2 - 2 - 3
3 - 1 - 2
4 - 0 - 1
5 - 2 - 3
6 - 3 - 4
*/
```

- Remarque sur la méthode BigM.

Jobshop : définition de prédicat

- Dans cet exemple de planification (jobshop), un cas de définition de prédicat personnel est donné.
- On doit organiser des tâches qui doivent s'exécuter les une après les autres.

```

int : taille;                                     %taille du problème
array[1..taille, 1..taille] of int : duree;      %durées des taches
int : total = sum(i, j in 1..taille)(duree[i, j]); %total durées
array[1..taille, 1..taille] of var 0..total : debut; %temps debut
var 0..total : fin;                               %total temps fin

predicate no_overlap(var int : debut1, int : duree1, var int : debut2, int : duree2)=
    debut1 + duree1 <= debut2  $\vee$  debut2 + duree2 <= debut1;

constraint
    forall(i in 1..taille) (
        forall(j in 1..taille-1) (
            debut[i, j] + duree[i, j] <= debut[i, j+1]
        )
         $\wedge$ 
        debut[i, taille] + duree[i, taille] <= fin
         $\wedge$ 
        forall(j, k in 1..taille where j < k) (
            no_overlap(debut[j, i], duree[j, i], debut[k, i], duree[k, i])
        )
    );

solve minimize fin;

```


Jobshop : définition de prédicat (suite)

```

output["fin = ", show(fin)++"\n"]
++ ["\nDebuts\n"]
++ [show(debut[i,j])++" " | i,j in 1..taille]
++ ["\nDurees\n"]
++ [show(duree[i,j])++" " | i,j in 1..taille]
++ ["\nFins\n"]
++ [show(debut[i,j]+duree[i,j])++" " | i,j in 1..taille];

```

% DATA

```

taille = 2;
duree = [
  2,5|
  3,4|
  |];

```

```

%-----
/*Une solution
fin = 11

```

```

Debuts
0 2 2 7
Durees
2 5 3 4
Fins
2 7 5 11
*/

```

- Remarque sur "predicate".

Exemple : où est le zèbre ?

Ce problème est énoncé par les faits suivants :

- 5 maisons consécutives dans une rue, de couleurs différentes, habitées par des hommes de nationalités différentes, chacun avec un animal différent et boit une boisson différente et fume des cigarettes de marques différentes.
- Les 5 nationalités : anglais, espagnole, ukrainien, japonais, norvégien.
- On sait que :
 - La maison verte est à droite de la maison de couleur ivoire.
 - Celui qui fume des Winstons élève des escargots.
 - On boit du lait dans la maison du milieu.
 - Le Norvégien est dans la première maison à gauche.
 - Le fumeur de Chesterfield habite à côté de celle avec un Renard.
 - Dans la maison verte, on boit du café.
 - Celui qui fume des Kools est dans la maison jaune.
 - Le fumeur de Lucky-Strike boit du jus d'orange.
 - Le Norvégien habite à côté de la maison bleue.

→ Qui boit de l'eau ? Où est le zèbre ?

.../ ...

Exemple : où est le zèbre ? (suite)

Le tableau suivant résume les informations, les complète et aide dans la résolution (raisonnement logique).

couleur	nationalité	animal	boisson	cigarettes
rouge	anglais	?	?	?
?	espagnole	chien	?	?
?	ukrainien	?	thé	?
?	japonnais	?	?	Chesterfield
?	norvégien	?	?	?

couleurs = {rouge, jaune, ivoire, verte, bleue}

animaux = {chien, renard, zèbre, escargots, cheval }

boissons = { eau minérale, lait, café, thé, jeu d'orange }

cigarettes = {Lucky-Strike, Kools, Winstons, Chesterfield, Marlboro }

Les '?' représentent les inconnues.

Qui possède le zèbre et qui boit de l'eau (minérale) ?

Solution (Zèbre)*

- Une idée : numéroter les *nationalités* de 1 à 5 :
 - ➔ Anglais=3 veut dire : le 3e maison est habitée par un anglais.
- Comment exprimer "la maison verte est à droite de la maison blanche"?
 - Une bonne idée : numéroter les maisons de 1 à 5 :
 - Et la première maison est (supposée) à gauche.
 - ➔ *Le Norvégien est dans la première maison à gauche* donne Norvégien=2
 - ➔ Jaune=2 veut dire : la 2e maison est jaune.
 - ➔ *La maison verte est à droite de la maison rouge* : Verte = Rouge +1
- ☞ On prendra un tableau de 5 entiers (1..5) pour chaque caractéristique.
- On utilise 25 variables : 5 par catégories (voir l'énoncé).
 - Le domaine de chaque variable est de 1 à 5 (les maisons).
 - Pour les animaux : on connaît chien, escargots, renard, cheval, ..
 - Pour les boissons : on a thé, café, lait, jus, ..
 - Imposer "tous différents" aux 5 paquets de 5 variables.

Solution (Zèbre)* (suite)

Le tableau précédent devient une collection de 5 tableaux chacun de 5 éléments.

couleur	nationalité	animal	boisson	cigarettes
rouge	anglais	?	?	?
?	espagnole	chien	?	?
?	ukrainien	?	thé	?
?	japonnais	?	?	Chesterfield
?	norvégien	?	?	?

- Chaque ligne : `array[1..5] of 1..5 : Couleurs;`

→ idem pour les 4 autres caractéristique.

- ☞ Pour chaque tableau, tous les éléments doivent être différents.

- Rappel des domaines des variables :

couleurs = {anglais, espagnole, ukrainien, japonais, norvégien}

couleurs = {rouge, jaune, ivoire, verte, bleue}

animaux = {chien, renard, zèbre, escargots, cheval }

boissons = {eau, lait, café, thé, jeu d'orange }

cigarettes = {Lucky-Strike, Kools, Winstons, Chesterfield, Marlboro }

Une solution Minizinc

```

include "globals.mzn";
set of 1..5: intervalle = 1..5;      % Indice 1 2 3 4 5
array[intervalle] of var intervalle: Nat; % Nat : norv esp ukr jap angl
array[intervalle] of var intervalle: Coul; % Colors : jaune bleu rouge vert ivoire
array[intervalle] of var intervalle: Anim; % Animaux : chien renard zebre escargot cheval
array[intervalle] of var intervalle: Boiss; % Boissons : eau lait cafe the orange
array[intervalle] of var intervalle: Cig; % Cigarette : lucky kools winston chester marlbo

% on décide que les maisons vont de gauche à droite et la première habité e par le Norvégien
constraint
  all_different(Nat)
  /\ all_different(Coul)
  /\ all_different(Anim)
  /\ all_different(Boiss)
  /\ all_different(Cig)
;

constraint
  forall(i in intervalle) (
    (Cig[i]=3 -> Anim[i]=4) % winston (3) => escargot (4)
    /\ % cheterfield (4) à coté du renard (2):
    (((i < 5) -> (Cig[i]=4 -> Anim[i+1]=2)) /\ ((i > 1) -> (Cig[i]=4 -> Anim[i-1]=2)))
  );

% 2e forme d'itération (avec garde) :
constraint % vert(5) à droite ivoire(5) :
  forall([Coul[i]=5 -> Coul[i-1]=4 | i in intervalle where i < 5])
;

% 3e forme d'itération
constraint % vert(4) boit café(3)
  forall([Coul[i]=4 -> Boiss[i]=3 | i in intervalle]) ;

```

Une solution Minizinc (suite)

```

constraint
  forall(i in intervalle) (
    (Cig[i]=2 -> Coul[i]=1) % kools et jaune
    /\ (Cig[i]=1 -> Boiss[i]=5) % lucky et jus orange
  );

constraint
  Boiss[3]=2 % 3e maison (du milieu) boit du lait (2)
  /\ Nat[1]=1 % 1ere maison = norvégien
  ;

solve satisfy;

output ["National = ", show(Nat) , "\n"]
++ ["Couleur = ", show(Coul) , "\n"]
++ ["Cigarette = ", show(Cig) , "\n"]
++ ["Boisson = ", show(Boiss) , "\n"]
++ ["Animal = ", show(Anim) , "\n"]
;

/*
National = [1, 5, 4, 3, 2]
Couleur = [4, 2, 1, 3, 5]
Cigarette = [5, 3, 2, 1, 4]
Boisson = [3, 4, 2, 5, 1]
Animal = [5, 4, 3, 2, 1]
*/

```

Une solution Minizinc (suite)

Une autre solution (sans les implications) pour une variante du problème :

```
include "globals.mzn";

set of 1..5: nat = 1..5; % Nationalities: Norwegian, Dane, Briton, Swede, German
set of 1..5: col = 1..5; % Colors: yellow blue red green white
set of 1..5: pet = 1..5; % Pets: cat bird dog horse fish
set of 1..5: drink = 1..5; % Drinks: coffee tea milk juice water
set of 1..5: smoke = 1..5; % Smokes: Dunhill Marlboro Pall - Mall Blumaster Prince

array[nat] of var nat: Tnat;
array[col] of var col: Tcol;
array[pet] of var pet: Tpet;
array[drink] of var drink: Tdrink;
array[smoke] of var smoke: Tsmoke;

array[nat] of string: nationalities = ["Norwegian", "Dane", "Briton", "Swede", "German"];

solve satisfy;

constraint
  all_different(Tnat) /\
  all_different(Tcol) /\
  all_different(Tpet) /\
  all_different(Tdrink) /\
  all_different(Tsmoke) /\

  Tnat[3] = Tcol[3] /\
  Tnat[4] = Tpet[3] /\
  Tnat[2] = Tdrink[2] /\
  Tcol[4] + 1 = Tcol[5] /\
  Tcol[4] = Tdrink[1] /\
  Tsmoke[3] = Tpet[2] /\
```


Une solution Minizinc (suite)

```

Tdrink[3] = 3 ∧
Tcol[1] = Tsmoke[1] ∧
Tnat[1] = 1 ∧
(Tsmoke[2] = Tpet[1]+1 ∨ Tsmoke[2] = Tpet[1]-1) ∧
(Tpet[4] = Tsmoke[1]+1 ∨ Tpet[4] = Tsmoke[1]-1) ∧
Tsmoke[4] = Tdrink[4] ∧
Tcol[2] = 2 ∧
Tnat[5] = Tsmoke[5] ∧
(Tsmoke[2] = Tdrink[5] + 1 ∨ Tsmoke[2] = Tdrink[5] - 1)
;

output [
  "Tnat: ", show(Tnat), "\n",
  "Tcol: ", show(Tcol), "\n",
  "Tpet: ", show(Tpet), "\n",
  "Tdrink: ", show(Tdrink), "\n",
  "Tsmoke: ", show(Tsmoke), "\n"
] ++
["The " ++ show(nationalities[fix(Tnat[Tpet[5]])]) ++ " owns the fish\n"]++["\n"];

/*
Solution
Tnat: [1, 2, 3, 5, 4]
Tcol: [1, 2, 3, 4, 5]
Tpet: [1, 3, 5, 2, 4]
Tdrink: [4, 2, 3, 5, 1]
Tsmoke: [1, 2, 3, 5, 4]
The German owns the fish
*/

```

Exemple Monnaie

- Soit un *montant* (par exemple $M = 28$) et un stock de petites pièces $P_i, i \in 1..k$ (par exemple, $S = [1, 7, 23]$). Proposer une séquence $E = [P_1, P_2, \dots, P_l]$ où $P_j \in S$ telle que $\sum_1^l P_j = M$ avec l minimal.
- Par exemple, pour $M = 28$, on peut proposer différentes S :
 - $S = [1, 1, \dots, 1]$ (28 fois 1)
 - $S = [23, 1, 1, 1, 1]$ (5 pièces)
 - $S = [7, 7, 7, 7]$ (4 pièces) : solution **optimale**.
- ☞ Pour faciliter l'écriture du modèle, mieux vaut considérer la solution sous la forme d'une séquence E' de la même taille que S où $E' = [V_1, V_2, \dots, V_k]$ avec $V_i \geq 0$ représente le nombre d'occurrence de la *ième* pièce de S qu'on veut utiliser dans la solution. Par exemple, la solution optimale ci dessus sera associée à $E' = [0, 4, 0]$: on utilise 4 pièces de 7. C'est donc $\text{somme}(E')$ qui doit être minimisée.

Exercice Paysans (3)

- 3 agriculteurs, 4 outils
- chacun donne ses préférences, un jour par outil.
- organiser en minimisant le nombre de jours
- Un outil utilisé par l'un ne peut être utilisé le même jour par un autre.
- Un agriculteur utilise chaque outil toute une journée.

Les préférences (H=taille_haies, T=tronçonneuse, B=boucheuse, M=motoculteur):

Modeste :< H, T, B, M >

Paul :< T, M, H, B >

claud :< M, T, B, H >

- Une solution possible :

	H	M	B	T
M	j1	j4	j3	j2
P	j3	j2	j5	j1
C	j5	j1	j4	j3

Remarques sur les exercices

- Les exercices suivants sont donnés avec leur barème (entre parenthèses).
- Respecter les consignes suivantes concernant vos choix :
 - Au moins deux exercices de 5 points (et plus) doivent être traités.
 - Au moins deux exercices dont les barèmes sont entre 3 et 4 points doivent être traités.
 - Ne pas traiter plus de 3 exercices à seulement à 1 point.
 - Le reste est laissé à votre choix.

Exercice BIP (trivial, 1)

- Donner des conseils à cet entrepreneur pour un problème de type entrepôt (Modélisation BIP) :

Une entreprise souhaite construire une nouvelle usine à Lyon ou à Grenoble (ou les deux).

- Un seul entrepôt mais là où on aura construit l'usine.
- Le bénéfice et le cout de chaque est donné dans la table ci-dessous.
- L'objectif est de maximiser les bénéfices.

Question oui/non	variable de décision	Bénéfices	couts
usine à Lyon	X_1	9 millions	6 millions
usine à Grenoble	X_2	5	3
entrepôt à Lyon	X_3	6	5
entrepôt à Grenoble	X_4	4	2

Capitaux disponibles max

10 millions.

- Combien faut-il des capitaux pour un rapport maximal $\frac{\text{gains}}{\text{couts}}$

Exercice : Gâteaux (3)

- Un pâtissier a deux recettes :

Tarte à la banane

250g de farine
2 bananes,
75g sucre
100g beurre

Tarte au chocolat

200g de farine
75g cacao,
150g sucre
150g beurre

- Il dispose de 4kg de farine, 6 bananes, 2kg de sucre, 500g de beurre et 500g de cacao.
- Il vend les tartes de chocolat à **4,5 euros** et 4,0 euros pour les tartes à la banane.
- Maximiser ses bénéfices.

Exercice : Régression (3)

- On dispose de quelques points et on souhaite procéder à une régression linéaire pour ces points. La droite représentative doit minimiser l'erreur absolue (mais linéaire).
- Proposer une solution permettant de trouver les coefficients de la droite $y = f(x) = ax + b$ et donner l'erreur.
- Les points x_i, y_i dans $f(x) = y$ sont (p.ex. pour $n = 13$ points):

1.0,	3.6,
1.5,	3.7,
2.0,	3.9,
2.5,	4.3,
3.0,	4.5,
3.5,	4.72,
4.0,	5.1,
4.2,	5.2,
5.0,	5.7,
5.8,	6.0,
6.0,	6.2,
6.7,	6.4,
7.2,	6.7

- Indication 1 : minimiser une somme d'erreurs linéaires car les contraintes quadratique (donc non linéaires) ne sont pas prises en charge par *Minzinc*.

Exercice : Régression (3) (suite)

- Indication 2 : la fonction $abs(.)$ s'appliquera ici que si un domaine est précisé pour a, b (on peut prendre p.ex. $[-100.0.. +100.0]$).

☞ Au lieu d'écrire

$$err_i \geq abs(ax_i + b - y_i)$$

où a, b sont les coefficient de la droite

à trouver, on peut plutôt utiliser :

$$err_i \geq ax_i + b - y_i \text{ ET } err_i \geq -(ax_i + b - y_i)$$

☞ Il y a 3 raisons pour cette notation de l'erreur :

- 1- Puisque l'erreur devra être minimisée, l'expression de l'erreur devra être $err \geq \text{expression}$ faute de quoi (si avec \leq) le minimum de l'erreur sera null !
- 2- Dans la forme "standard" d'un Programme Mathématique, lorsque l'objectif est minimisé, les opérateurs de comparaisons seront \geq (quitte à transformer le programme pour épouser cette forme). Ici, la forme standard est respectée.

- 3- Pour supprimer abs dans $err \geq |Expr|$:

si $Expr \geq 0$ alors $err \geq Expr$

Sinon on aura $-Expr \geq 0$ et donc $err \geq -Expr$

- Indication 3 : pour un cas à 2 dimensions, l'expression des coefficients a et b par la méthode moindre carrée (minimisation des carrés des erreurs) permet une meilleure approximation :

$$\hat{a} = \frac{n \cdot \sum_1^n x_i y_i - \sum_1^n x_i \cdot \sum_1^n y_i}{n \cdot \sum_1^n (x_i)^2 - \left(\sum_1^n x_i \right)^2}$$

$$\hat{b} = \frac{\sum_1^n (x_i)^2 \cdot \sum_1^n y_i - \sum_1^n x_i \cdot \sum_1^n x_i y_i}{n \cdot \sum_1^n (x_i)^2 - \left(\sum_1^n x_i \right)^2}$$

Exercice : Régression (3) (suite)

- Comme il a été indiqué en cours, si vous ne disposez pas de la fonction $abs(.)$ (ce qui n'est pas notre cas ici!), si $abs(.)$ ne s'applique pas aux réels ou enfin si vous devez optimiser son utilisation, vous pouvez utiliser la technique suivante pour la transformer.

L'expression qui peut remplacer $abs(.)$:

- Soit la contrainte $abs(f(x)) \geq m$
- On pose (z est un booléen, U et L sont les bornes Sup/Inf de m) :
 - Si $z = 1$ alors on a $f(x) \geq 0$ et donc $U \geq f(x) \geq m$ (I)
 - Si $z = 0$ alors on a $f(x) < 0$ et donc $-m \geq f(x) \geq L$ (II)
 - Et donc : $z * (U + m) - m \geq f(x) \geq L + z * (m - L)$ (III)
- Dans le cas présent, $m = err_i$ est l'erreur individuelle pour x_i
 et $f(x) = ax + b - y$.
- On peut raisonnablement poser $L = 0.0$ et $U = 1.0$
 (mais une valeur plus grande pour U p.ex. 100 est possible, cf. Big_M)
- Dans ce cas, (III) devient $m \geq f(x) \geq -m$

Exercice voyage (2)

Exercice Tourné (un exemple BIP)

- Pour une prospection, un étudiant veut visiter les campus de trois universités en Rhône-Alpes pendant un voyage à partir de "Lyon" et retour.

Les trois Univ sont situées dans les villes "St-Etienne", "Valence" et "Grenoble" et l'étudiant veut visiter chaque ville universitaire une seule fois tout en faisant l'aller-retour **le plus court possible**.

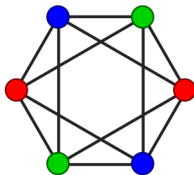
- La table suivante donne les distances entre les villes :

Villes	Ville 1	Ville 2	Ville 3	Ville 4
	Lyon	St-Etienne	Valence	Grenoble
Lyon	0	26	34	78
St-Etienne	26	0	18	52
Valence	34	18	0	51
Grenoble	78	52	51	0

- Proposer une solution Minizinc.

Exercice coloration (2)

- Soit le graphe suivant à colorer de sorte que 2 noeuds connectés ne reçoivent pas la même couleur.



- Proposer une solution Minizinc minimisant le nombre de couleurs utilisées.

Exercice fret (1)

Minimisation du cout total.

Transporter 42 tonnes de fret avec 8 camions de 4 volumes différents :

Type	Nb dispo	Capacité (tonnes)	cout par camion
1	3	7	90
2	3	5	60
3	3	4	50
4	3	3	40

Modèle :

- Variables : $x_{i=1..4}$: nombre de chaque camion utilisé (i=le type : 1..4)

$$\min \quad 90x_1 + 60x_2 + 50x_3 + 40x_4$$

$$s.t. \quad 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42$$

$$x_1 + x_2 + x_3 + x_4 \leq 8$$

$$7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 \quad \text{appelé "contrainte couverture sac-à-dos"}$$

$$x_1 + x_2 + x_3 + x_4 \leq 8 \quad \text{appelé "contrainte remplissage sac-à-dos"}$$

- Proposer une solution Minizinc.

Exercice Jobshop (5)

- Une série de tâches avec pour chacune une date au plus tôt ($Rel=release$) où la tâche peut commencer, une durée de la tâche et une date de livraison (qu'on va dépasser pour certaines).
- Si la date de livraison d'une tâche est dépassée, on tient compte de la différence (pénalité en vue!); par contre, si une tâche est prête avant sa date de livraison, le dépassement = 0 (car il nous faut tout finir pour livrer).
- On a une seule machine : donc tâche après tâche passent sur celle-ci.
→ Minimiser les dépassement.
- Dans tous les cas, la somme des durées (30) dépasse la date de livraison (22). On nous accordera peut être la différence (8), mais pas plus!
- Les données du problème sont ci-dessous :

Exercice Jobshop (5) (suite)

rel : Le moment où une tâche est disponible pour la machine
dur : Durée de la tâche
due : Le moment où la tâche doit être terminée

<i>JOBS</i>	<i>rel</i>	<i>dur</i>	<i>due</i>
A	2	5	10
B	5	6	21
C	4	8	15
D	0	4	10
E	0	2	5
F	8	3	15
G	9	2	22

/ Une solution à trouver :*

<i>Task</i>	<i>Rel</i>	<i>Dur</i>	<i>Due</i>	<i>Start</i>	<i>Finish</i>	<i>Pastdue (dépassement)</i>
A	2	5	10	2	7	0
B	5	6	21	14	20	0
C	4	8	15	22	30	15
D	0	4	10	7	11	1
E	0	2	5	0	2	0
F	8	3	15	11	14	0
G	9	2	22	20	22	0

**/*

Exercice : Jésuites (5)

Trouver une solution pour le problème suivant.

Énoncé :

- Il y a 7 jésuites dans une maison (abbaye)
- Leurs taches sont divisées en 4 groupes :
Cuisine, SdB, Partie commune et Poubelles.
- La poubelle est la seule tâche qui a besoin d'une personne;
les autres tâches ont besoin de 2 personnes.
- Chaque personne a besoin de faire une tache 2 fois sur les 7 semaines (et celle de la poubelle une fois)
- Trouver une affectation par semaine (sur les 7 semaines)

Exercice : Jésuites (5) (suite)

Modélisation :

- Toute personne doit avoir un nouveau partenaire chaque semaine (donc ne pas reconduire une équipe de 2 d'une semaine à l'autre)
- Personne ne doit avoir plus d'une tâche par semaine
- On utilise un tableau à 4-1 dimensions de bool avec S : les semaines, T : tâches, V : volontaires (les jours J : pas besoin !)
 $Cube[S : 1..7, T : 1..4, V : 1..7] :: 0..1$
- $Cube[S, T, V] = 1$ veut dire que le volontaire V fera la tâche T la semaine S
 → Créer cette cube dimension par dimension pour pouvoir faire facilement des sommes!
- Une personnes pour la poubelle, 2 pour les autres.
- Faire la somme des ligne dans chaque semaine.

../..

Exercice : Jésuites (5) (suite)

Les contraintes :

Pour tout $S : 1..7$

pour tout $T :$

si $T = 4 \rightarrow \text{somme}(\text{Cube}[S,T])=1$ Poubelle

si $T \neq 4 \rightarrow \text{somme}(\text{Cube}[S,T])=2$ Autres

- Toute personne doit faire une tâche 2 fois (sur les 7 semaines) :

Pour tout $V : 1..7$

pour tout $T :$

pour tout $S, S', J \mid S \neq S'$

si $T = 4 \rightarrow \text{Cube}[S,T,V] + \text{Cube}[S',T,V]=1$ Poubelle

si $T \neq 4 \rightarrow \text{Cube}[S,J,V] + \text{Cube}[S',T,V]=2$ Autres

- Chaque personne doit avoir un nouveau partenaire chaque semaine (donc ne pas reconduire une équipe de 2 d'une semaine à l'autre)

d'où $S \neq S'$

- Personne ne doit avoir plus d'une tâche par semaine

Rappel : $\text{Cube}[S, T, V] = 1$ veut dire que le volontaire V fera la tâche T la semaine S

Pour tout S somme de chaque colonne = 1

Exercice : Jésuites (5) (suite)

Exemple de solution :

```
Semaine : 1
Volontaire:1 2 3 4 5 6 7
Kitchen : - - - - X X
Bathroom: - - - X X - -
Commons : X X - - - -
Trash : - - X - - -
```

```
Semaine : 2
Volontaire:1 2 3 4 5 6 7
Kitchen : - X X - - -
Bathroom: - - - X X -
Commons : - - - X - - X
Trash : X - - - - -
```

```
Semaine : 3
Volontaire:1 2 3 4 5 6 7
Kitchen : X - - X - -
Bathroom: - - X - - X
Commons : - X - X - -
Trash : - - - - X -
```

```
Semaine : 4
Volontaire:1 2 3 4 5 6 7
Kitchen : - - X X - -
Bathroom: - X - - - X
Commons : X - - - X -
Trash : - - - X - -
```

```
Semaine : 5
Volontaire:1 2 3 4 5 6 7
Kitchen : - X - - X - -
```

Exercice : Jésuites (5) (suite)

Bathroom: X - - X - - -
Commons : - - X - - X -
Trash : - - - - - X

Semaine : 6

Volontaire: 1 2 3 4 5 6 7
Kitchen : - - - X - X -
Bathroom: X - X - - - -
Commons : - - - - X - X
Trash : - X - - - - -

Semaine : 7

Volontaire: 1 2 3 4 5 6 7
Kitchen : X - - - - X
Bathroom: - X - - - X -
Commons : - - X - X - -

Exercice : Jésuites (5) (suite)

Et par paires :

Pair: (1,2)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -
 Bathroom: - - - - -
 Commons : X - - - -
 Trash : - - - - -

Pair: (1,3)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -
 Bathroom: - - - - X -
 Commons : - - - - -
 Trash : - - - - -

Pair: (1,4)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -
 Bathroom: - - - X - -
 Commons : - - - - -
 Trash : - - - - -

Pair: (1,5)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - X - - -
 Bathroom: - - - - -
 Commons : - - - - -
 Trash : - - - - -

Pair: (1,6)
 Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -

Exercice : Jésuites (5) (suite)

Bathroom : - - - - -
 Commons : - - - X - - -
 Trash : - - - - -

Pair: (1,7)

Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - - X
 Bathroom : - - - - -
 Commons : - - - - -
 Trash : - - - - -

Pair: (2,3)

Semaine : 1 2 3 4 5 6 7
 Kitchen : - X - - - -
 Bathroom : - - - - -
 Commons : - - - - -
 Trash : - - - - -

Pair: (2,4)

Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - -
 Bathroom : - - - - -
 Commons : - - X - - -
 Trash : - - - - -

Pair: (2,5)

Semaine : 1 2 3 4 5 6 7
 Kitchen : - - - - X -
 Bathroom : - - - - -
 Commons : - - - - -
 Trash : - - - - -

Pair: (2,6)

Semaine : 1 2 3 4 5 6 7

Exercice : Jésuites (5) (suite)

```

Kitchen : - - - - -
Bathroom: - - - - - X
Commons : - - - - -
Trash   : - - - - -

```

```

Pair: (2,7)
Semaine : 1 2 3 4 5 6 7
Kitchen  : - - - - -
Bathroom: - - - X - -
Commons  : - - - - -
Trash    : - - - - -

```

```

Pair: (3,4)
Semaine : 1 2 3 4 5 6 7
Kitchen  : - - - X - -
Bathroom: - - - - -
Commons  : - - - - -
Trash    : - - - - -

```

```

Pair: (3,5)
Semaine : 1 2 3 4 5 6 7
Kitchen  : - - - - -
Bathroom: - - - - -
Commons  : - - - - - X
Trash    : - - - - -

```

```

Pair: (3,6)
Semaine : 1 2 3 4 5 6 7
Kitchen  : - - - - -
Bathroom: - - - - -
Commons  : - - - - X -
Trash    : - - - - -

```

Exercice : Jésuites (5) (suite)

Pair: (3,7)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: - - X - - -

Commons : - - - - -

Trash : - - - - -

Pair: (4,5)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: X - - - - -

Commons : - - - - -

Trash : - - - - -

Pair: (4,6)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - - X -

Bathroom: - - - - -

Commons : - - - - -

Trash : - - - - -

Pair: (4,7)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: - - - - -

Commons : - X - - - -

Trash : - - - - -

Pair: (5,6)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: - X - - - -

Commons : - - - - -

Exercice : Jésuites (5) (suite)

Trash : - - - - -

Pair: (5,7)

Semaine : 1 2 3 4 5 6 7

Kitchen : - - - - -

Bathroom: - - - - -

Commons : - - - - - X -

Trash : - - - - -

Pair: (6,7)

Semaine : 1 2 3 4 5 6 7

Kitchen : X - - - - -

Bathroom: - - - - -

Commons : - - - - -

Trash : - - - - -

Exercice : Ateliers véhicules (5)

Planification de chaîne de production de 4 ateliers A, B, C, D

- A a une capacité de production de 5 véhicules/heure/homme
- B a une capacité de production de 3 V/h/h
- C a une capacité de production de 2 V/h/h
- D a une capacité de production de 3 V/h/h
- ➔ Tout véhicule a besoin de passer devant les 4 ateliers (dans cet ordre).
- Les ressources : on a 4 opérateurs Operateur1 .. Operateur4
 - Operateur1 est opérationnel entre 8h-13h
 - Operateur2 est opérationnel entre 8h-13h
 - Operateur3 est opérationnel entre 9h-13h
 - Operateur4 est opérationnel entre 10h-15h
- A 8h, 30 véhicules sont en attente devant l'atelier A, aucun devant les autres
- 👉 **Variantes** : des véhicules de modèles différents, avec options différentes
Ateliers / Opérateurs mono / multi compétences, ...

Exercice : Ateliers véhicules (5) (suite)

- *Proposer un emploi du temps heure par heure précisant l'affectation des ouvriers aux ateliers tout en **maximisant** le total de véhicules passés devant les 4 ateliers.*

Exercice : Découpe papier (2)

Du livre *Understading and using Linear programming*, page 26.

- Trouver une solution pour le problème suivant.
- A paper mill manufactures rolls of paper of a standard width 3 meters. But customers want to buy paper rolls of shorter width, and the mill has to cut such rolls from the 3 m rolls. One 3 m roll can be cut, for instance, into two rolls 93 cm wide, one roll of width 108 cm, and a rest of 6 cm (which goes to waste). Let us consider an order of
 - 97 rolls of width 135 cm,
 - 610 rolls of width 108 cm,
 - 395 rolls of width 93 cm, and
 - 211 rolls of width 42 cm.

What is the smallest number of 3 m rolls that have to be cut in order to satisfy this order, and how should they be cut?

Exercice Tournée (contrainte element, 2)

Organisation de tournée.

- Dans un problème de tournée, un contrôleur visite des sites :
 - Il visite le site S le jour J .
 - Il ne visite chaque site qu'une fois;
 - Il peut visiter 2 sites consécutifs en 2 jours consécutifs ou espacés d'au plus 1 jour.
- **Modélisation :**
 - Variables : site S_i , l'ensemble donne $Z = \text{Liste des sites ordonnée}$
 - Domaine : $S_i \in 1..7$ (les 7 jours).
 - Contraintes :
 - Visiter chaque site une seule fois : $\text{alldifferent}(Z)$
 - Espacement des visites : pour toute paire de sites S_i et S_j consécutifs :

$$S_i = S_j + X, I \in 1..4, \text{element}(I, [-1, -2, 1, 2], X)$$
- ☞ Quid si : si lundi travaillé, repos mercredi (et vice versa)

Exercice Bin Packing (3)

- Étant donné un ensemble d'articles $I = \{1, \dots, m\}$ avec un poids $w[i] > 0$, le problème d'emballage de bac (Bin Packing, BPP) est d'emballer les articles dans des bacs de c capacité de telle façon que le nombre de bacs utilisés soit minimal.
- Par exemple :
 - Soit des petites boites de poids $< 1, 2, 3, 4, 5, 6, 7 >$
 - Soit des grandes boites de capacité 20.
 - Emballer les petite boites en minimisant le nombre de grandes boites utilisées.
- Donner l'esquisse d'un modèle général et transformer votre programme en modèle + data (si ce n'est pas le cas).
 - Puis emballer 3 séries de petites boites. Minimiser les grandes !
- ☞ Il s'agit d'envisager un seul emballage des ces 3 séries en un seul envoi.

Bonus : Tournoi à organiser (5)

Un organisateur sportif prévoit un tournoi avec 8 équipes [Helmut Simonis-2009] :

- chaque équipe joue contre toutes les autres équipes une seule fois.
 - le tournoi se joue sur 7 jours,
 - chaque équipe joue tous les jours (des 7),
 - les matchs sont prévus sur 7 sites, et
 - chaque équipe doit jouer dans chaque site exactement une fois.
-
- Dans le cadre d'un accord avec la télévision, certains matchs sont pré-organisés.
 - On peut soit fixer le match entre deux équipes particulières à une date déterminée et un site déterminé,
 - Ou seulement décider par avance qu'une certaine équipe doit jouer un jour donné en un lieu donné.
 - **L'objectif** est de déterminer le calendrier, de sorte que toutes les contraintes soient satisfaites.

Bonus : Tournoi à organiser (5) (suite)

Expressions des exigences (contraintes) :

→ permettent d'envisager différentes contraintes / solutions :

- Il y a 8 équipes, 7 jours et 7 sites
- Chaque équipe joue chaque autre équipe une fois exactement
- Chaque équipe joue 7 matchs (redondant)
- Chaque équipe joue exactement une fois dans chaque site
- Chaque équipe joue chaque jour exactement une fois
- Un match se compose de 2 différentes équipes
- Il y a 4 matchs chaque jour (redondant)
- Il y a 4 matchs à chaque emplacement (redondants)
- Dans n'importe quel site, il y a au plus un match à la fois

☞ Selon les choix, on peut proposer différentes idées de solution ../..

Bonus : Tournoi à organiser (5) (suite)

- **Idée 1 :** utiliser une matrice $Jour \times match$ (7×4)
 - Chaque cellule contient 2 variables (deux équipes)
 - Contrainte : toute équipe joue une seule fois chaque jour (*alldifferent*)
 - Les colonnes n'auront pas de signification
 - Les sites non représentés : comment faire ?
 - Comment dire : chaque équipe joue avec une autre une seule fois.
- **Idée 2 :** variables binaires pour exprimer : équipe i joue en site j le jour k .
 - Matrice à 3 dimensions
 - Chaque équipe joue une seule fois chaque jour
 - Chaque équipe joue une seule fois dans chaque site
 - Un match a 2 différentes équipes ? (variables auxiliaires nécessaires).
 - Chaque pair d'équipe se rencontrent une seule fois ? (vars auxiliaires).
- **Idée 3 :** variables booléennes pour exprimer : équipe i rencontre équipe j en site k le jour l .
 - $3136 = 8 * 8 * 7 * 7$ variables
 - Toutes les contraintes sont linéaires

Bonus : Tournoi à organiser (5) (suite)

- Idée 4 : chaque équipe joue contre une autre exactement une fois :
 - $7 * 4$ matchs à organiser (28 variables)
 - Toutes les variables différentes (pas de match le même jour, le même lieu)
 - Par construction, chaque équipe jouera 7 matchs
 - Comment dire : chaque équipe jouera une fois par jour ?
 - Comment dire : chaque équipe jouera une fois par site ?
- Cette idée donnera :

	City 1	City 2	City 3	City 4	City 5	City 6	City 7
Day 1	1	2	3	4	5	6	7
Day 2	8	9	10	11	12	13	14
Day 3	15	16	17	18	19	20	21
Day 4	22	23	24	25	26	27	28
Day 5	29	30	31	32	33	34	35
Day 6	36	37	38	39	40	41	42
Day 7	43	44	45	46	47	48	49

Bonus : Tournoi à organiser (5) (suite)

- Jour 1 : valeurs 1..7
- 4 variables prendront une des ces valeurs (1..7)
- Jour 2 : 8..15, &c.
- Une contrainte par jour
- Exactement 4 variables prendront leur valeurs dans la ligne correspondantes
- 7 de ces contraintes (car 7 lignes).

- Le site 1 correspond aux valeurs 1,8,15, 22, ...
- 4 variables prendront une des ces valeurs
- Site 2 correspond à 2, 9,16, ...
- Une contrainte par site
- Exactement 4 variables prendront leur valeurs dans l'ensemble correspondant
- 7 de ces contraintes sur chacune des 28 variables.

- Choisir les variables qui correspondent à l'équipe i
 - Exactement une de ces variables prendront un evaleur dans 1..7
- De même pour les autres jours
- De même pour les autres matches
 - 56 contraintes sur chacune des 7 variables
- De même pour les équipes et sites :
 - 56 autres contraintes

	City 1	City 2	City 3	City 4	City 5	City 6	City 7
Day 1	1	2	3	4	5	6	7
Day 2	8	9	10	11	12	13	14
Day 3	15	16	17	18	19	20	21
Day 4	22	23	24	25	26	27	28
Day 5	29	30	31	32	33	34	35
Day 6	36	37	38	39	40	41	42
Day 7	43	44	45	46	47	48	49

Bilan de l'idée 4 :

- 28 variables,
- Un *alldifferent*
- 7 contraintes sur toutes les variables (pour les jours)
- 7 contraintes sur toutes les variables (pour les sites)
- 56 contraintes sur 7 variables (pour les jours)
- 56 contraintes sur 7 variables (pour les sites)
- ...et on n'a pas encore fini!

Bonus : Tournoi à organiser (5) (suite)

- Idée 5 (reprise en CP) :
 - une matrice carré de $Jour \times Site = 49$ de matchs (couples d'équipes $[A,B]$)
 - Chaque cellule contient un match (couples d'équipes)
 - Comment éviter les symétries ($[A,B]$ vs $[B,A]$)
 - Utiliser $[0,0]$ pour l'absence de match
 - Une cellule contient $[0,0]$ ou $[A,B]$, $A \neq B \neq 0$.
 - Plus facile : chaque ligne / colonne contient un match une seule fois.
 - Attention à *alldifferent* (pour les zéros)
 - Comment exprimer : chaque paire ordonnée apparaît une seule fois ?
 - Comment exprimer : chaque équipe joue une fois par jour / site ?
- Et du côté des contraintes non-domaine-fini ?
- 👉 Mettre mon ex de choix SD n-reines / complexité

Une solution CP

- Le tableau initial (avec les matchs pré organisés), puis une solution :

	City 1	City 2	City 3	City 4	City 5	City 6	City 7
Day 1		8			7, 5		
Day 2	2	1, 5					
Day 3	7		8				
Day 4					2	5	1
Day 5	8					1	
Day 6				5, 4			
Day 7	4				1, 3		

	City 1	City 2	City 3	City 4	City 5	City 6	City 7
Day 1		6, 8		1, 2	5, 7		3, 4
Day 2	2, 3	1, 5			4, 8	6, 7	
Day 3	1, 7	2, 4	3, 8				5, 6
Day 4			4, 7		2, 6	3, 5	1, 8
Day 5	5, 8			3, 6		1, 4	2, 7
Day 6		3, 7	1, 6	4, 5		2, 8	
Day 7	4, 6		2, 5	7, 8	1, 3		

Une solution CP (suite)

Une spécification (applicable à cette classe de problèmes) :

Placez numéros 1 à 8 dans les cellules de la matrice de sorte que :

- dans chaque rangée et dans chaque colonne figure chaque numéro d'équipe ($\in 1..8$) exactement une fois, et
 - chaque cellule contient soit pas de chiffres, soit un couple de chiffres ($\in 1..8$, = un match) différents, et
 - chaque couple de chiffres (un match) apparaît dans seulement une cellule.
- On peut envisager la structure de données suivante :
 Pour 8 équipes ($E1..E8$) :
 - M : une matrice 7×7
 - Lignes : les jours ($Ji : i = 1..7$)
 - Colonnes : les Sites ($Si, i = 1..7$)
 - Une case : vide ou $[Eu, Ev]$: les deux équipes qui jouent

Une solution CP (suite)

- **Choix des variables :**

Une matrice carré \mathbf{M} de $7 \times 7 = 49$ de couples d'équipes ($Jours \times Site$)

→ Chaque couple dans une cellule $M_{ij}, i, j = 1..7$

- **Domaines :**

49 cellules contenant chacune un couple $[Eu, Ev], u, v \in 0..8, u \neq v$,

☞ Pour une cellule vide (pas de match $jour \times site$), on utilisera le couple $[0, 0]$.

- **Contraintes :**

Pour simplifier et éviter les solutions symétriques, on décide que dans un couple non vide (donc : $\neq [0, 0]$) $[Eu, Ev] : u < v$.

- Cel_i est autorisée à contenir soit $[0, 0]$ soit $[Eu, Ev], Eu \neq 0, Ev \neq 0$
- Déf : $[Eu, Ev] \neq [Eu', Ev']$ si $u \neq u'$ ou $v \neq v'$
- Pour chaque ligne de la matrice M contenant 7 cellules $Cel_{i=1..7}$ (représentant tous les matchs d'un jour) :
 - Si $Cel_i = [Eu, Ev] \neq [0, 0]$ alors $Cel_i \neq Cel_{k=1..7, i \neq k}$
 - Que toutes les équipes soient impliquées dans $Cel_{i=1..7}$

Une solution CP (suite)

- Pour chaque colonne de la matrice M contenant 7 cellules $Cel_{j=1..7}$ (représentant tous les matchs d'un site) :
 - Si $Cel_j = [Eu, Ev] \neq [0, 0]$ alors $Cel_j \neq Cel_{k=1..7, k \neq j}$
 - Que toutes les équipes soient impliquées dans $Cel_{j=1..7}$

- Il y a beaucoup de solutions.

- Une solution (autre) :

$[[0, 0], [0, 0], [0, 0], [1, 2], [3, 4], [5, 6], [7, 8]]$

$[[7, 8], [3, 4], [5, 6], [0, 0], [0, 0], [0, 0], [1, 2]]$

$[[5, 6], [0, 0], [0, 0], [3, 4], [8, 7], [1, 2], [0, 0]]$

$[[0, 0], [7, 2], [1, 8], [0, 0], [0, 0], [3, 4], [5, 6]]$

$[[3, 4], [6, 8], [0, 0], [5, 7], [1, 2], [0, 0], [0, 0]]$

$[[0, 0], [1, 5], [2, 7], [6, 8], [0, 0], [0, 0], [3, 4]]$

$[[1, 2], [0, 0], [3, 4], [0, 0], [5, 6], [7, 8], [0, 0]]$

- Aspects déclaratifs de la CP.

PERT avec ressources : 5 (2 si cumulative utilisée)

- Minimiser le temps total (*makespan*) de réalisation des tâches sans deadline.
- On a :
 - 5 tâches t_1, \dots, t_5
avec les durées respectives 3, 3, 3, 5, 5
et les ressources nécessaires requises 3, 3, 3, 2, 2
- ☞ Minimiser la date de la fin des travaux.
- Il n'y a pas de contrainte de précédence.
- Une modélisation CP est donnée en page suivante.

Indication : pour la gestion des ressources limitées, on peut envisager (au besoin) de contrôler chaque unité du temps.

PERT avec ressources : 5 (2 si cumulative utilisée) (suite)

Une modélisation CP (avec *cumulative*) :

$\min Z$

s.t. $\text{cumulative}((t_1, \dots, t_5),$
 $(3, 3, 3, 5, 5), (3, 3, 3, 2, 2), 7)$

$Z \geq t_1 + 3$

.....

$Z \geq t_5 + 2$

Avec :

(t_1, \dots, t_5) : date de début des tâches

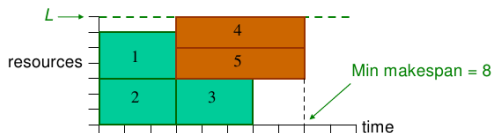
$(3, 3, 3, 5, 5)$: durées des tâches

$(3, 3, 3, 2, 2)$ ressources utilisées par les tâches

→ Ne jamais dépasser la limite des ressources 7

→ On a le total des durées = 8 (vs la somme des durées = 19)

☞ *cumulative* existe en Minizinc.



Sur les docks (5)

Du manuel OPL (un langage d'optimisation) :

Les données du problème :

- Planifier 34 containers sur un bateau en un temps minimum (min *makespan*).
 - Le chargement de chaque container nécessite un certain temps et un certain nombre d'ouvriers (cf. tableau).
 - On dispose de 8 ouvriers.
- ☞ **Pour simplifier** : considérer seulement une partie de cette table.

Container	Durée	Nb. Ouvriers
1	3	4
2	4	4
3	4	3
4	6	4
5	5	5
6	2	5
7	3	4
8	4	3
9	3	4
10	2	8
11	3	4
12	2	5
13	1	4
14	5	3
15	2	3
16	3	3
17	2	6

Container	Durée	Nb. Ouvriers
18	2	7
19	1	4
20	1	4
21	1	4
22	2	4
23	4	7
24	5	8
25	2	8
26	1	3
27	1	3
28	2	6
29	1	8
30	3	3
31	2	3
32	1	3
33	2	3
34	2	3

Sur les docks (5) (suite)

- Les contraintes de précédence :

1 → 2,4	11 → 13	22 → 23
2 → 3	12 → 13	23 → 24
3 → 5,7	13 → 15,16	24 → 25
4 → 5	14 → 15	25 → 26,30,31,32
5 → 6	15 → 18	26 → 27
6 → 8	16 → 17	27 → 28
7 → 8	17 → 18	28 → 29
8 → 9	18 → 19	30 → 28
9 → 10	18 → 20,21	31 → 28
9 → 14	19 → 23	32 → 33
10 → 11	20 → 23	33 → 34
10 → 12	21 → 22	

$t_i \rightarrow t_j$: t_j commence après la fin de t_i .

- Le modèle CP :

$\min \quad Z$

$s.t. \quad \text{cumulative}((t_1, \dots, t_{34}),$
 $(3, 4, 2, \dots, 2), (4, 4, \dots, 3), 8)$

$Z \geq t_1 + 3$

$Z \geq t_2 + 4$

.....

$t_2 \geq t_1 + 3$

$t_4 \geq t_1 + 3$

.....

Infirmières (5)

- Soit 4 infirmières travaillant pendant des périodes (shift) de 8 heures.
- Une infirmière travaille au plus un shift par jour.
- Une infirmière travaille au moins 5 jours la semaine.
- Même planification pour chaque semaine.
- Chaque shift contient deux infirmières (différentes) par semaine.
- Une infirmière ne peut pas travailler 2 shifts différents sur 2 jours consécutifs.
- Une infirmière qui travaille shift 2 ou 3 doit faire de même au moins deux jours d'affiler.
- Exemple d'assignation des infirmières aux shifts (créneaux longs).

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Shift 1	A	B	A	A	A	A	A
Shift 2	C	C	C	B	B	B	B
Shift 3	D	D	D	D	C	C	D

- Exemple d'assignation des shifts aux infirmières.

Infirmières (5) (suite)

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Nurse A	1	0	1	1	1	1	1
Nurse B	0	1	0	2	2	2	2
Nurse C	2	2	2	0	3	3	0
Nurse D	3	3	3	3	0	0	3

☞ Donner un tableau des shifts en respectant les contraintes.

→ Il y a plusieurs solutions possibles.

Restaurant (5)

- Résoudre le problème du restaurateur :
 - il veut minimiser son nombre d'employés sachant qu'un employé travaille 5 jours d'affilée puis a deux jours de repos.
 - Le personnel requis par jour de la semaine est donné par le tableau :

Jour	L	M	M	J	V	S	D
Nombre	14	13	15	16	19	18	11

- Vous donnerez le nombre total d'employés et la répartition par jour de travail.

Financement (3)

Résoudre le problème de financement ci-dessous :

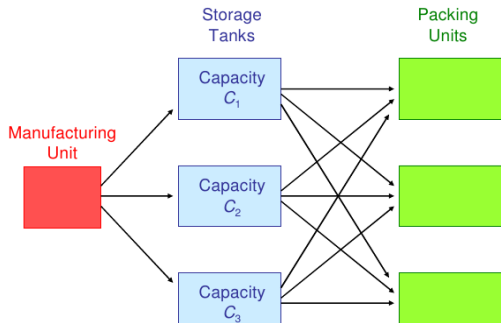
- Placer 1000 euros sur 6 ans de la manière optimale sachant que :
 - la caisse d'épargne rapporte 5% par an et immobilise le capital un an,
 - l'obligation 1 rapporte 12% à l'échéance si on le choisit la première année sinon elle rapporte 11% et immobilise le capital deux ans,
 - l'obligation 2 rapporte 18% à l'échéance et immobilise le capital trois ans,
 - l'obligation 3 rapporte 24% à l'échéance et immobilise le capital quatre ans,
 - L'obligation 2 est disponible tous les ans sauf l'année 3,
 - l'obligation 2 n'est pas disponible l'année 1,
 - et l'obligation 3 n'est disponible que l'année 1.
- Vous donnerez le gain attendu et la répartition par type de placement chaque année
- Calculez le taux moyen de rendement et comparez-le au taux de la caisse d'épargne.

Voyageur (3)

- Un voyageur peut rapporter pour les vendre divers objets.
 - Malheureusement la compagnie aérienne limite la charge qu'il peut emporter dans son sac à dos à 40kg ce qui lui laisse uniquement 14kg (car il a déjà ses propres affaires).
 - Le premier objet pèse 5kg et se revend avec un bénéfice de 8 euros,
 - Le second pèse 7kg et rapporte 11 euros,
 - Le troisième pèse 4kg et rapporte 6 euros
 - Et le dernier pèse 3kg et rapporte 4 euros.
- (a) Introduire une variable x_i à valeur dans $\{0, 1\}$ qui signifie prendre ou pas l'objet i , et modéliser le problème sous forme d'un problème de programmation linéaire en relâchant la contrainte $x_i \in \{0, 1\}$ en $0 \leq x_i \leq 1$.
- (b) Résoudre le problème de programmation linéaire.
- (c) Pour chaque solution non entière $x_i = n_i$, résoudre les deux problèmes obtenus en posant $x_i \leq n_i$ et $x_i \geq n_i$.
- Continuer jusqu'à trouver la solution du problème de départ.

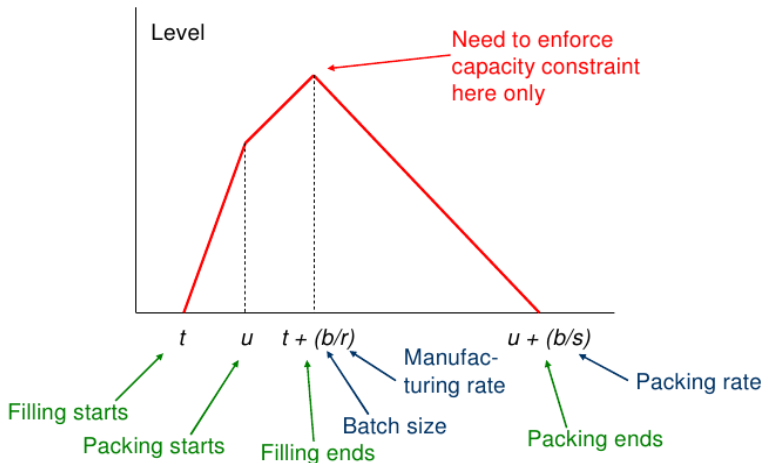
Stockage (Cumulative, 3)

- Planification de production avec stockage intermédiaire :



- Voir la page suivante.
- 👉 Donner le minimum de la date de l'achèvement !

Stockage (Cumulative, 3) (suite)



Stockage (Cumulative, 3) (suite)

- Une formulation avec des contraintes.

$$\min \quad T \quad \leftarrow (\text{makespan}) A \text{ minimiser}$$

$$s.t. \quad T \geq u_j + \frac{b_j}{s_j}, \forall j$$

$$t_j \geq R_j, \forall j \quad \leftarrow \text{temps fin tâche } t_j$$

$$\text{cumulative}(t, v, e, m) \quad \leftarrow m \text{ tankers de stockage}$$

$$v_i = u_i + \frac{b_i}{s_i} - t_i, \forall i \quad \leftarrow \text{duree des tâches } t_i$$

$$b_i \left(1 - \frac{s_i}{r_i}\right) + s_i u_i \leq C_i, \forall i \quad \leftarrow \text{capacite du tanker } i$$

$$\text{cumulative}(u, \left(\frac{b_1}{s_1}, \dots, \frac{b_n}{s_n}\right), e, p) \quad \leftarrow p \text{ units de paquetage}$$

$$u_j \geq t_j \geq 0 \quad e = (1, \dots, 1)$$

Contraintes Globales de Minizinc

(voir par famille plus loin)

- **Convention** en Minizinc

- \$T ou \$U représente tout type. \$T peut être de type int, float, etc.
- Dans l'expression

function set of \$U: array_union(array [\$T] of set of \$U: x),

\$U et \$T peuvent être différents types mais dans la fonction

function set of \$T: 'intersect'(set of \$T: x, set of \$T: y)

tous les \$ sont du même type.

→ Différents noms de variables après \$ veulent dire qu'ils peuvent être de type différents. Alors que le même nom après \$ imposent d'avoir le même type.

- Exemple:

function set of float: array_union(array [int] of set of float: x)

Et

function set of int: 'intersect'(set of int: x, set of int: y).

- array [\$T] est en quelque sorte spécial et veut dire que le tableau est d'une dimension quelconque.

i.e array [int], array [int,int] ou array [int,int,int,int,int] etc.

Donc array [\$T] of set of \$U veut dire qu'on peut avoir un tableau de dimension \$T, par exemple [int,int] (une matrice). Ce tableau est rempli d'ensembles de n'importe quel type. Par exemple, *sets of integers* comme dans 1,4,7,145.

⚠ *var int* et *int* sont différents. *int* est un paramètre entier tandis que *var int* est une variable de décision de type entier.

Contraintes Globales de Minizinc (suite)

La liste alphabétique des contraintes globales de MiniZinc.

- **alldifferent** (array[int] of var int: x)
alldifferent(array[int] of var set of int: x)
→ Constrains the array of objects x to be all different. Also available by the name all_different.
- alldifferent_except_0(array[int] of var int: x)
→ Constrains the elements of the array x to be all different except those elements that are assigned the value 0.
- all_disjoint(array[int] of var set of int: x)
→ Ensures that every pair of sets in the array x is disjoint.
- all_equal(array[int] of var int: x)
all_equal(array[int] of var set of int: x)
→ Constrains the array of objects x to have the same value.
- **among** (var int: n, array[int] of var int: x, set of int: v)
→ Requires exactly n variables in x to take one of the values in v.
- at_least(int: n, array[int] of var int: x, int: v)
at_least(int: n, array[int] of var set of int: x, set of int: v)
→ Requires at least n variables in x to take the value v. Also available by the name atleast.
- at_most(int: n, array[int] of var int: x, int: v)
at_most(int: n, array[int] of var set of int: x, set of int: v)
→ Requires at most n variables in x to take the value v. Also available by the name atmost.
- at_most1(array[int] of var set of int: s)
→ Requires that each pair of sets in s overlap in at most one element.
→ Also available by the name atmost1.
- **bin_packing** (int: c, array[int] of var int: bin, array[int] of int: w)
→ Requires that each item i be put into bin bin[i] such that the sum of the weights of each item, w[i], in each bin does not exceed the capacity c.
→ Aborts if an item has a negative weight or if the capacity is negative.
→ Aborts if the index sets of bin and w are not identical.

Contraintes Globales de Minizinc (suite)

- `bin_packing_capa`(array[int] of int: c, array[int] of var int: bin, array[int] of int:w)
 - Requires that each item i be put into bin $\text{bin}[i]$ such that the sum of the weights of each item, $w[i]$, in each bin b does not exceed the capacity $c[b]$. Aborts if an item has negative weight. Aborts if the index sets of bin and w are not identical.
- `bin_backing_load`(array[int] of var int: l, array[int] of var int: bin, array[int] of int: w)
 - Requires that each item i be put into bin $\text{bin}[i]$ such that the sum of the weights of each item, $w[i]$, in each bin b is equal to the load $l[b]$. Aborts if an item has negative weight. Aborts if the index sets of bin and w are not identical.
- `circuit` [array[int] of var int: x)
 - Constrains the elements of x to define a circuit where $x[i] = j$ mean that j is the successor of i .
- `count_eq` (array[int] of var int: x, var int: y, var int: c)
 - Constrains c to be the number of occurrences of y in x . Also available by the name `count`.
- `count_geq`(array[int] of var int: x, var int: y, var int: c)
 - Constrains c to greater than or equal to the number of occurrences of y in x .
- `count_gt`(array[int] of var int: x, var int: y, var int: c)
 - Constrains c to strictly greater than the number of occurrences of y in x .
- `count_leq`(array[int] of var int: x, var int: y, var int: c)
 - Constrains c to less than or equal to the number of occurrences of y in x .
- `count_lt`(array[int] of var int: x, var int: y, var int: c)
 - Constrains c to strictly less than the number of occurrences of y in x .
- `count_neq`(array[int] of var int: x, var int: y, var int: c)
 - Constrains c to not be the number of occurrences of y in x .
- `cumulative` (array[int] of var int: s, array[int] of var int: d, array[int] of var int: r, var int: b)
 - Requires that a set of tasks given by start times s , durations d , and resource requirements r , never require more than a global resource bound b at any one time. Aborts if s , d , and r do not have identical index sets. Aborts if a duration or resource requirement is negative.

Contraintes Globales de Minizinc (suite)

- decreasing(array[int] of var bool: x)
 decreasing(array[int] of var float: x)
 decreasing(array[int] of var int: x)
 decreasing(array[int] of var set of int: x)
 → Requires that the array x is in (non-strictly) decreasing order.
- diffn** (array[int] of var int: x, array[int] of var int: y, array[int] of var int: dx, array[int] of var int: dy)
 → Constrains rectangles, given by their origins x,y and sizes dx,dy, to be non-overlapping.
- disjoint** (var set of int: s, var set of int: t)
 → Requires that sets s and t do not intersect.
- distribute** (array[int] of var int: card, array[int] of var int: value, array[int] of var int: base)
 → Requires that card[i] is the number of occurrences of value[i] in base. In this implementation the values in value need not be distinct. Aborts if card and value do not have identical index sets.
- element** (var int: i, array[int] of var bool: x, var bool: y)
 element(var int: i, array[int] of var float: x, var float: y)
 element(var int: i, array[int] of var int: x, var int: y)
 element(var int: i, array[int] of var set of int: x, var set of int: y)
 → The same as $x[i] = y$. That is, y is the ith element of the array x.
- exactly** (int: n, array[int] of var int: x, int: v)
 exactly(int: n, array[int] of var set of int: x, set of int: v)
 → Requires exactly n variables in x to take the value v.
- global_cardinality** (array[int] of var int: x, array[int] of int: cover, array[int] of var int: counts)
 → Requires that the number of occurrences of cover[i] in x is counts[i]. Aborts if cover and counts do not have identical index sets.
- global_cardinality_closed** (array[int] of var int: x, array[int] of int: cover, array[int] of var int: counts)
 → Requires that the number of occurrences of cover[i] in x is counts[i]. The elements of x must take their values from cover. Aborts if cover and counts do not have identical index sets.
- global_cardinality_low_up** (array[int] of var int: x, array[int] of int: cover, array[int] of int: lb, array[int] of int: ub)
 → Requires that for all i, the value cover[i] appears at least lb[i] and at most ub[i] times in the array x.

Contraintes Globales de Minizinc (suite)

- `global_cardinality_low_up_closed(array[int] of var int: x, array[int] of int: cover, array[int] of int: lb, array[int] of int: ub)`
→ Requires that for all i , the value `cover[i]` appears at least `lb[i]` and at most `ub[i]` times in the array `x`. The elements of `x` must take their values from `cover`.
- `increasing(array[int] of var bool: x)`
`increasing(array[int] of var float: x)`
`increasing(array[int] of var int: x)`
`increasing(array[int] of var set of int: x)`
→ Requires that the array `x` is in (non-strictly) increasing order.

Channeling (assignment)

- `int_set_channel(array[int] of var int: x, array[int] of var set of int: y)`
→ Requires that $x[i] = j$ if and only if $i \in y[j]$.
- `inverse(array[int] of var int: f, array[int] of var int: invf)`
→ **Assign**
→
→ Constrains two arrays to represent inverse functions of each other. All the values in each array must be within the index set of the other array.
- `inverse_set(array[int] of var set of int: f, array[int] of var set of int: invf)`
→ Constrains the two arrays `f` and `invf` so that $j \in f[i]$ if and only if $i \in invf[j]$. All the values in each array's sets must be within the index set of the other array.
- `lex_greater(array[int] of var bool: x, array[int] of var bool: y)`
`lex_greater(array[int] of var float: x, array[int] of var float: y)`
`lex_greater(array[int] of var int: x, array[int] of var int: y)`
`lex_greater(array[int] of var set of int: x, array[int] of var set of int: y)`
→ Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.

Contraintes Globales de Minizinc (suite)

- `lex_greatereq(array[int] of var bool: x, array[int] of var bool: y)`
`lex_greatereq(array[int] of var float: x, array[int] of var float: y)`
`lex_greatereq(array[int] of var int: x, array[int] of var int: y)`
`lex_greatereq(array[int] of var set of int: x, array[int] of var set of int: y)`
 → Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.
- `lex_less(array[int] of var bool: x, array[int] of var bool: y)`
`lex_less(array[int] of var float: x, array[int] of var float: y)`
`lex_less(array[int] of var int: x, array[int] of var int: y)`
`lex_less(array[int] of var set of int: x, array[int] of var set of int: y)`
 → Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- `lex_lesseq(array[int] of var bool: x, array[int] of var bool: y)`
`lex_lesseq(array[int] of var float: x, array[int] of var float: y)`
`lex_lesseq(array[int] of var int: x, array[int] of var int: y)`
`lex_lesseq(array[int] of var set of int: x, array[int] of var set of int: y)`
 → Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- `lex2(array[int, int] of var int: x)`
 → Require adjacent rows and adjacent columns in the the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns may be equal.
- link_set_to_booleans** (var set of int: `s`, array[int] of var bool: `b`)
 → The array of booleans `b` is the characteristic representation of the set `s`. Aborts if the index set of `b` is not a superset of the possible values of `s`.
- maximum** (var int: `m`, array[int] of var int: `x`)
`maximum(var float: m, array[int] of var float: x)`
 → Constrains `m` to be the maximum of the values in `x`. (The array `x` must have at least one element.)

Contraintes Globales de Minizinc (suite)

- member** (array[int] of var bool: x, var bool: y)
 member(array[int] of var float: x, var float: y)
 member(array[int] of var int: x, var int: y)
 member(array[int] of var set of int: x, var set of int: y)
 member(var set of int: x, var int: y)
 → Requires that y occurs in the array or set x.
- minimum** (var float: m, array[int] of var float: x)
 minimum(var int: m, array[int] of var int: x)
 → Constrains m to be the minimum of the values in x. (The array x must have at least one element.)
- nvalue** (var int: n, array[int] of var int: x)
 → Requires that the number of distinct values in x is n.
- partition_set** (array[int] of var set of int: s, set of int: universe)
 → Partitions universe into disjoint sets.
- range** (array[int] of var int: x, var set of int: s, var set of int: t)
 → Requires that the image of function x (represented as an array) on set of values s is t. Aborts if $ub(s)$ is not a subset of the index set of x.
- regular** (array[int] of var int: x, int: Q, int: S, array[int,int] of int: d, int: q0, set of int: F)
 → The sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $\langle 1..Q, 1..S \rangle$ to $0..Q$) and initial state q0 (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state. Aborts if $Q < 1$. Aborts if $S < 1$. Aborts if the transition function d is not in $[1..Q, 1..s]$. Aborts if the start state, q0, is not in $1..Q$. Aborts if F is not a subset of $1..Q$.
- roots** (array[int] of var int: x, var set of int: s, var set of int: t)
 → Requires that $x[i] \in t$ for all $i \in s$. Aborts if $ub(s)$ is not a subset of the index set of x.
- sliding_sum** (int: low, int: up, int: seq, array[int] of var int: vs)
 → Requires that in each subsequence $vs[i], \dots, vs[i + seq - 1]$ the sum of the values belongs to the interval [low, up].
- sort** (array[int] of var int: x, array[int] of var int: y)
 → Requires that the multiset of values in x is the same as the multiset of values in y but y is in sorted order. Aborts if the cardinality of the index sets of x and y is not equal.

Contraintes Globales de Minizinc (suite)

- `strict_lex2(array[int, int] of var int: x)`
→ Require adjacent rows and adjacent columns in the the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns cannot be equal.
- `subcircuit (array[int] of var int: x)`
→ Constrains the elements of `x` to define a subcircuit where `x[i] = j` means that `j` is the successor of `i` and `x[i] = i` means that `i` is not in the circuit.
- `sum_pred (var int: i, array[int] of set of int: sets, array[int] of int: c, var int: s)`
→ Requires that the sum of `c[i1]...c[iN]` equals `s`, where `i1...iN` are the elements of the `ith` set in `sets`.
→ This constraint is usually named `sum`, but using that would conflict with the MiniZinc built-in function of the same name.
- `table (array[int] of var bool: x, array[int, int] of bool: t)`
`table(array[int] of var int: x, array[int, int] of int: t)`
→ Represents the constraint $x \in t$ where we consider each row in `t` to be a tuple and `t` as a set of tuples. Aborts if the second dimension of `t` does not equal the number of variables in `x`. The default decomposition of this constraint cannot be flattened if it occurs in a reified context.
- `value_precede (int: s, int: t, array[int] of var int: x)`
`value_precede(int: s, int: t, array[int] of var set of int: x)`
→ Requires that `s` precede `t` in the array `x`. For integer variables this constraint requires that if an element of `x` is equal to `t`, then another element of `x` with a lower index is equal to `s`. For set variables this constraint requires that if an element of `x` contains `t` but not `s`, then another element of `x` with lower index contains `s` but not `t`.
- `value_precede_chain(array[int] of int: c, array[int] of var int: x)`
`value_precede_chain(array[int] of int: c, array[int] of var set of int: x)`
→ Requires that the `value_precede` constraint is true for every pair of adjacent integers in `c` in the array `x`.

Contraintes globales par famille

La liste alphabétique des contraintes globales de MiniZinc.

* All-Different and related constraints

- predicate `all_different(array [int] of var int: x)`
Constrain the array of integers `x` to be all different.
- predicate `all_different(array [int] of var set of int: x)`
Constrain the array of sets of integers `x` to be all different.
- predicate `all_disjoint(array [int] of var set of int: S)`
Constrain the array of sets of integers `S` to be pairwise disjoint.
- predicate `all_equal(array [int] of var int: x)`
Constrain the array of integers `x` to be all equal
- predicate `all_equal(array [int] of var set of int: x)`
Constrain the array of sets of integers `x` to be all different
- predicate `alldifferent_except_0(array [int] of var int: vs)`
Constrain the array of integers `vs` to be all different except those elements that are assigned the value 0.
- function `var int: nvalue(array [int] of var int: x)`
Returns the number of distinct values in `x`.
- predicate `nvalue(var int: n, array [int] of var int: x)`
Requires that the number of distinct values in `x` is `n`.
- predicate `symmetric_all_different(array [int] of var int: x)`
Requires the array of integers `x` to be all different, and for all i , $x[i] = j \implies x[j] = i$.

Contraintes globales par famille (suite)

* Lexicographic constraints

- predicate `lex2(array [int,int] of var int: x)`
Require adjacent rows and adjacent columns in the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns may be equal.
- predicate `lex_greater(array [int] of var bool: x, array [int] of var bool: y)`
Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greater(array [int] of var int: x, array [int] of var int: y)`
Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greater(array [int] of var float: x, array [int] of var float: y)`
Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greater(array [int] of var set of int: x, array [int] of var set of int: y)`
Requires that the array `x` is strictly lexicographically greater than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greatereq(array [int] of var bool: x, array [int] of var bool: y)`
Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greatereq(array [int] of var int: x, array [int] of var int: y)`
Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greatereq(array [int] of var float: x, array [int] of var float: y)`
Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_greatereq(array [int] of var set of int: x, array [int] of var set of int: y)`
Requires that the array `x` is lexicographically greater than or equal to array `y`. Compares them from first to last element, regardless of indices.

Contraintes globales par famille (suite)

- predicate `lex_less(array [int] of var bool: x, array [int] of var bool: y)`
Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_less(array [int] of var int: x, array [int] of var int: y)`
Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_less(array [int] of var float: x, array [int] of var float: y)`
Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_less(array [int] of var set of int: x, array [int] of var set of int: y)`
Requires that the array `x` is strictly lexicographically less than array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_lesseq(array [int] of var bool: x, array [int] of var bool: y)`
Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_lesseq(array [int] of var float: x, array [int] of var float: y)`
Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_lesseq(array [int] of var int: x, array [int] of var int: y)`
Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `lex_lesseq(array [int] of var set of int: x, array [int] of var set of int: y)`
Requires that the array `x` is lexicographically less than or equal to array `y`. Compares them from first to last element, regardless of indices.
- predicate `strict_lex2(array [int,int] of var int: x)`
Require adjacent rows and adjacent columns in the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns cannot be equal.

Contraintes globales par famille (suite)

En particulier les précédences de valeurs

- predicate `value_precede(int: s, int: t, array [int] of var int: x)`
Requires that `s` precede `t` in the array `x`.
Precedence means that if any element of `x` is equal to `t`, then another element of `x` with a lower index is equal to `s`.
- predicate `value_precede(int: s, int: t, array [int] of var set of int: x)`
Requires that `s` precede `t` in the array `x`.
Precedence means that if an element of `x` contains `t` but not `s`, then another element of `x` with lower index contains `s` but not `t`.
- predicate `value_precede_chain(array [int] of int: c, array [int] of var int: x)`
Requires that `c[i]` precedes `c[i + 1]` in the array `x`.
Precedence means that if any element of `x` is equal to `c[i + 1]`, then another element of `x` with a lower index is equal to `c[i]`.
- predicate `value_precede_chain(array [int] of int: c, array [int] of var set of int: x)`
Requires that `c[i]` precedes `c[i + 1]` in the array `x`.
Precedence means that if an element of `x` contains `c[i + 1]` but not `c[i]`, then another element of `x` with lower index contains `c[i]` but not `c[i + 1]`.

Contraintes globales par famille (suite)

* **Sorting constraints** **Et in/decreasing ponctuel**

- function `array [int] of var int: arg_sort(array [int] of var int: x)`
Returns the permutation p which causes x to be in sorted order hence $x[p[i]] \leq x[p[i + 1]]$. The permutation is the stable sort hence $x[p[i]] = x[p[i + 1]] \implies p[i] < p[i + 1]$.
- function `array [int] of var int: arg_sort(array [int] of var float: x)`
Returns the permutation p which causes x to be in sorted order hence $x[p[i]] \leq x[p[i + 1]]$. The permutation is the stable sort hence $x[p[i]] = x[p[i + 1]] \implies p[i] < p[i + 1]$.
- predicate `arg_sort(array [int] of var int: x, array [int] of var int: p)`
Constrains p to be the permutation which causes x to be in sorted order hence $x[p[i]] \leq x[p[i + 1]]$. The permutation is the stable sort hence $x[p[i]] = x[p[i + 1]] \implies p[i] < p[i + 1]$.
- predicate `arg_sort(array [int] of var float: x, array [int] of var int: p)`
Constrains p to be the permutation which causes x to be in sorted order hence $x[p[i]] \leq x[p[i + 1]]$. The permutation is the stable sort hence $x[p[i]] = x[p[i + 1]] \implies p[i] < p[i + 1]$.
- predicate `decreasing(array [int] of var bool: x)`
Requires that the array x is in decreasing order (duplicates are allowed).
- predicate `decreasing(array [int] of var float: x)`
Requires that the array x is in decreasing order (duplicates are allowed).
- predicate `decreasing(array [int] of var int: x)`
Requires that the array x is in decreasing order (duplicates are allowed).
- predicate `decreasing(array [int] of var set of int: x)`
Requires that the array x is in decreasing order (duplicates are allowed).
- predicate `increasing(array [int] of var bool: x)`
Requires that the array x is in increasing order (duplicates are allowed).
- predicate `increasing(array [int] of var float: x)`
Requires that the array x is in increasing order (duplicates are allowed).

Contraintes globales par famille (suite)

- predicate **increasing**(array [int] of var int: x)
Requires that the array x is in **increasing order** (duplicates are allowed).
- predicate **increasing**(array [int] of var set of int: x)
Requires that the array x is in **increasing order** (duplicates are allowed).
- function array [int] of var int: **sort**(array [int] of var int: x)
Return a multiset of values that is the same as the multiset of values in x but in sorted order.
- predicate **sort**(array [int] of var int: x, array [int] of var int: y)
Requires that the multiset of values in x are the same as the multiset of values in y but y is in sorted order.

Contraintes globales par famille (suite)

* Channeling constraints (assignment)

- predicate `int_set_channel`(array [int] of var int: x, array [int] of var set of int: y)
Requires that array of int variables x and array of set variables y are related such that
 $(x[i] = j) \iff (i \text{ in } y[j])$.
- function array [int] of var int: `inverse`(array [int] of var int: f)
Given a function f represented as an array, return the inverse function.
- predicate `inverse`(array [int] of var int: f, array [int] of var int: invf)
Constrains two arrays of int variables, f and invf, to represent inverse functions.
All the values in each array must be within the index set of the other array.
- predicate `inverse_set`(array [int] of var set of int: f, array [int] of var set of int: invf)
Constrains two arrays of set of int variables, f and invf, so that a j in f[i] iff i in invf[j].
All the values in each array's sets must be within the index set of the other array.
- predicate `link_set_to_booleans`(var set of int: s, array [int] of var bool: b)
Constrain the array of Booleans b to be a representation of the set s: $i \text{ in } s \iff b[i]$.
The index set of b must be a superset of the possible values of s.

Contraintes globales par famille (suite)

* Counting constraints

- function var int: **among**(array [int] of var int: x, set of int: v)
Returns the number of variables in x that take one of the values in v.
- predicate **among**(var int: n, array [int] of var int: x, set of int: v)
Requires exactly n variables in x to take one of the values in v.
- predicate **at_least**(int: n, array [int] of var int: x, int: v)
Requires at least n variables in x to take the value v.
- predicate **at_least**(int: n, array [int] of var set of int: x, set of int: v)
Requires at least n variables in x to take the value v.
- predicate **at_most**(int: n, array [int] of var int: x, int: v)
Requires at most n variables in x to take the value v.
- predicate **at_most**(int: n, array [int] of var set of int: x, set of int: v)
Requires at most n variables in x to take the value v.
- predicate **at_most1**(array [int] of var set of int: s)
Requires that each pair of sets in s overlap in at most one element.
- function var int: **count**(array [int] of var int: x, var int: y)
Returns the number of occurrences of y in x.
- predicate **count**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be the number of occurrences of y in x.
- predicate **count_eq**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be the number of occurrences of y in x.
- predicate **count_geq**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be greater than or equal to the number of occurrences of y in x.
- predicate **count_gt**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be strictly greater than the number of occurrences of y in x.
- predicate **count_leq**(array [int] of var int: x, var int: y, var int: c)
Constrains c to be less than or equal to the number of occurrences of y in x.

Contraintes globales par famille (suite)

- predicate `count_lt`(array [int] of var int: x, var int: y, var int: c)
Constrains c to be strictly less than the number of occurrences of y in x.
- predicate `count_neq`(array [int] of var int: x, var int: y, var int: c)
Constrains c to be not equal to the number of occurrences of y in x.
- function array [int] of var int: `distribute`(array [int] of var int: value, array [int] of var int: base)
Returns an array of the number of occurrences of value[i] in base.
The values in value need not be distinct.
- predicate `distribute`(array [int] of var int: card, array [int] of var int: value, array [int] of var int: base)
Requires that card[i] is the number of occurrences of value[i] in base.
The values in value need not be distinct.
- predicate `exactly`(int: n, array [int] of var int: x, int: v)
Requires exactly n variables in x to take the value v.
- predicate `exactly`(int: n, array [int] of var set of int: x, set of int: v)
Requires exactly n variables in x to take the value v.
- function array [int] of var int: `global_cardinality`(array [int] of var int: x, array [int] of int: cover)
Returns the number of occurrences of cover[i] in x.
- predicate `global_cardinality`(array [int] of var int: x, array [int] of int: cover, array [int] of var int: counts)
Requires that the number of occurrences of cover[i] in x is counts[i].
- function array [int] of var int: `global_cardinality_closed`(array [int] of var int: x, array [int] of int: cover)
Returns an array with number of occurrences of i in x.
The elements of x must take their values from cover.
- predicate `global_cardinality_closed`(array [int] of var int: x, array [int] of int: cover, array [int] of var int: counts)
Requires that the number of occurrences of i in x is counts[i].
The elements of x must take their values from cover.

Contraintes globales par famille (suite)

- predicate `global_cardinality_low_up`(array [int] of var int: x, array [int] of int: cover,
array [int] of int: lbound, array [int] of int: ubound)
Requires that for all i, the value cover[i] appears at least lbound[i] and
at most ubound[i] times in the array x.
- predicate `global_cardinality_low_up_closed`(array [int] of var int: x, array [int] of int: cover,
array [int] of int: lbound, array [int] of int: ubound)
Requires that for all i, the value cover[i] appears at least lbound[i] and
at most ubound[i] times in the array x.
The elements of x must take their values from cover.

Contraintes globales par famille (suite)

* Packing constraints

- predicate `bin_packing`(int: c, array [int] of var int: bin, array [int] of int: w)
Requires that each item i with weight w[i], be put into bin[i] such that the sum of the weights of the items in each bin does not exceed the capacity c.

Assumptions: $\forall i, w[i] \geq 0, c \geq 0$

- predicate `bin_packing_capa`(array [int] of int: c, array [int] of var int: bin, array [int] of int: w)
Requires that each item i with weight w[i], be put into bin[i] such that the sum of the weights of the items in each bin b does not exceed the capacity c[b].

Assumptions: $\forall i, w[i] \geq 0, \forall b, c[b] \geq 0$

- function array [int] of var int: `bin_packing_load`(array [int] of var int: bin, array [int] of int: w)
Returns the load of each bin resulting from packing each item i with weight w[i] into bin[i], where the load is defined as the sum of the weights of the items in each bin.

Assumptions: $\forall i, w[i] \geq 0$

- predicate `bin_packing_load`(array [int] of var int: load, array [int] of var int: bin, array [int] of int: w)
Requires that each item i with weight w[i], be put into bin[i] such that the sum of the weights of the items in each bin b is equal to load[b].

Assumptions: $\forall i, w[i] \geq 0$

- predicate `diffn`(array [int] of var int: x, array [int] of var int: y, array [int] of var int: dx, array [int] of var int: dy)
Constrains rectangles i, given by their origins (x[i], y[i]) and sizes (dx[i], dy[i]), to be non-overlapping.
Zero-width rectangles can still not overlap with any other rectangle.

- predicate `diffn_k`(array [int,int] of var int: box_posn, array [int,int] of var int: box_size)
Constrains k-dimensional boxes to be non-overlapping.
For each box i and dimension j, box_posn[i, j] is the base position of the box in dimension j, and box_size[i, j] is the size in that dimension.
Boxes whose size is 0 in any dimension still cannot overlap with any other box.

Contraintes globales par famille (suite)

- predicate `diffn_nonstrict`(array [int] of var int: x, array [int] of var int: y, array [int] of var int: dx, array [int] of var int: dy)
 Constrains rectangles i, given by their origins (x[i], y[i]) and sizes (dx[i], dy[i]), to be non-overlapping.
 Zero-width rectangles can be packed anywhere.
- predicate `diffn_nonstrict_k`(array [int,int] of var int: box_posn, array [int,int] of var int: box_size)
 Constrains k-dimensional boxes to be non-overlapping.
 For each box i and dimension j, box_posn[i, j] is the base position of the box in dimension j, and box_size[i, j] is the size in that dimension.
 Boxes whose size is 0 in at least one dimension can be packed anywhere.
- predicate `geost`(int: k, array [int,int] of int: rect_size, array [int,int] of int: rect_offset, array [int] of set of int: shape, array [int,int] of var int: x, array [int] of var int: kind)
 A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap.

Parameters

- k: the number of dimensions
- rect_size: the size of each box in k dimensions
- rect_offset: the offset of each box from the base position in k dimensions
- shape: the set of rectangles defining the i-th shape.
 → Assumption: Each pair of boxes in a shape must not overlap.
- x: the base position of each object. x[i,j] is the position of object i in dimension j.
- kind: the shape used by each object.

Contraintes globales par famille (suite)

- predicate `geost_bb`(int: k, array [int,int] of int: rect_size, array [int,int] of int: rect_offset, array [int] of set of int: shape, array [int,int] of var int: x, array [int] of var int: kind, array [int] of var int: l, array [int] of var int: u)

A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap, and that all objects fit within a global k dimensional bounding box.

Parameters

- k: the number of dimensions
- rect_size: the size of each box in k dimensions
- rect_offset: the offset of each box from the base position in k dimensions
- shape: the set of rectangles defining the i-th shape.
→ Assumption: Each pair of boxes in a shape must not overlap.
- x: the base position of each object. $x[i,j]$ is the position of object i in dimension j.
- kind: the shape used by each object.
- l: is an array of lower bounds, $l[i]$ is the minimum bounding box for all objects in dimension i.
- u: is an array of upper bounds, $u[i]$ is the maximum bounding box for all objects in dimension i.
- predicate `geost_smallest_bb`(int: k, array [int,int] of int: rect_size, array [int,int] of int: rect_offset, array [int] of set of int: shape, array [int,int] of var int: x, array [int] of var int: kind, array [int] of var int: l, array [int] of var int: u)

A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap, and that all objects fit within a global k dimensional bounding box. In addition, it enforces that the bounding box is the smallest one containing all objects, i.e., each of the $2k$ boundaries is touched by at least by one object.

Parameters

- k: the number of dimensions
- rect_size: the size of each box in k dimensions
- rect_offset: the offset of each box from the base position in k dimensions
- shape: the set of rectangles defining the i-th shape.
→ Assumption: Each pair of boxes in a shape must not overlap.
- x: the base position of each object. $x[i,j]$ is the position of object i in dimension j.
- kind: the shape used by each object.
- l: is an array of lower bounds, $l[i]$ is the minimum bounding box for all objects in dimension i.
- u: is an array of upper bounds, $u[i]$ is the maximum bounding box for all objects in dimension i.

Contraintes globales par famille (suite)

- predicate `knapsack`(array [int] of int: w, array [int] of int: p, array [int] of var int: x, var int: W, var int: P)
Requires that items are packed in a knapsack with certain weight and profit restrictions.

Assumptions:

- Weights w and profits p must be non-negative
- w, p and x must have the same index sets

Parameters

- w: weight of each type of item
- p: profit of each type of item
- x: number of items of each type that are packed
- W: sum of sizes of all items in the knapsack
- P: sum of profits of all items in the knapsack

Contraintes globales par famille (suite)

* Scheduling constraints

- predicate **alternative**(var opt int: s0, var int: d0, array [int] of var opt int: s, array [int] of var int: d)
Alternative constraint for optional tasks.
Task (s0,d0) spans the optional tasks (s[i],d[i]) in the array arguments and at most one can occur
- predicate **cumulative**(array [int] of var opt int: s, array [int] of var int: d, array [int] of var int: r, var int: b)
Requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.
Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions: $\forall i, d[i] \geq 0 \wedge r[i] \geq 0$

- predicate **cumulative**(array [int] of var int: s, array [int] of var int: d, array [int] of var int: r, var int: b)
Requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.

Assumptions: $\forall i, d[i] \geq \wedge r[i] \geq 0$

- predicate **disjunctive**(array [int] of var opt int: s, array [int] of var int: d)
Requires that a set of tasks given by start times s and durations d do not overlap in time.
Tasks with duration 0 can be scheduled at any time, even in the middle of other tasks.
Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions: $\forall i, d[i] \geq 0$

- predicate **disjunctive**(array [int] of var int: s, array [int] of var int: d)
Requires that a set of tasks given by start times s and durations d do not overlap in time.
Tasks with duration 0 can be scheduled at any time, even in the middle of other tasks.

Assumptions: $\forall i, d[i] \geq 0$

Contraintes globales par famille (suite)

- predicate `disjunctive_strict`(array [int] of var opt int: s, array [int] of var int: d)
Requires that a set of tasks given by start times s and durations d do not overlap in time.
Tasks with duration 0 CANNOT be scheduled at any time, but only when no other task is running.
Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions: $\forall i, d[i] \geq 0$

- predicate `disjunctive_strict`(array [int] of var int: s, array [int] of var int: d)
Requires that a set of tasks given by start times s and durations d do not overlap in time.
Tasks with duration 0 CANNOT be scheduled at any time, but only when no other task is running.

Assumptions: $\forall i, d[i] \geq 0$

- predicate `span`(var opt int: s0, var int: d0, array [int] of var opt int: s, array [int] of var int: d)
Span constraint for optional tasks. Task (s0,d0) spans the **optional tasks** (s[i],d[i]) in the array arguments.

Contraintes globales par famille (suite)

* Extensional constraints (table, regular etc.)

- predicate **regular**(array [int] of var int: x, int: Q, int: S, array [int,int] of int: d, int: q0, set of int: F)
The sequence of values in array x (which must all be in the range 1..S) is accepted by the DFA of Q states with input 1..S and transition function d (which maps $(1..Q, 1..S) \rightarrow 0..Q$) and initial state q0 (which must be in 1..Q) and accepting states F (which all must be in 1..Q).
We reserve state 0 to be an always failing state.
- predicate **regular_nfa**(array [int] of var int: x, int: Q, int: S, array [int,int] of set of int: d, int: q0, set of int: F)
The sequence of values in array x (which must all be in the range 1..S) is accepted by the NFA of Q states with input 1..S and transition function d (which maps $(1..Q, 1..S) \rightarrow \text{set of } 1..Q$) and initial state q0 (which must be in 1..Q) and accepting states F (which all must be in 1..Q).
- predicate **table**(array [int] of var bool: x, array [int,int] of bool: t)
Represents the constraint x in t where we consider each row in t to be a tuple and t as a set of tuples.
- predicate **table**(array [int] of var int: x, array [int,int] of int: t)
Represents the constraint x in t where we consider each row in t to be a tuple and t as a set of tuples.

Installer GLPK

- Vous pourriez être intéressé par un autre outil de Programmation Mathématique : GLPK.

- **Installation sur Windows** : site

<http://sourceforge.net/projects/winglpk/files/winglpk/GLPK-4.47.1/>

- En décembre 2012, *winglpk-4.47.1*

- Interface graphique (GUI) pour glpk version Windows :

→ Récupérer l'outil "Gusek" sur le site

<http://gusek.sourceforge.net/gusek.html>

→ Décompresser, Installer, lancer Gusek

→ Une fenêtre s'ouvre (éditeur, résultats dans le cadre droit)

→ Dans les options, préciser que vous utilisez GMPL (.mod)

→ ouvrir ou créer un fichier (par ex., **pb.mod**)

→ L'icône "go" pour lancer glpsol sur le problème actuel

→ Pour le fichier de données, *"Tools > Use External .dat"*

→ A droite, vous pouvez taper les commandes :

`glpsol -model pb.mod [-data pb.dat -output pb.out]` (entre []=optionnel)

- Il y a aussi "GLPK Lab" pour Windows.

Installer GLPK (suite)

- **Installation sur Linux** : site

<http://ftp.gnu.org/gnu/glpk/glpk-4.47.tar.gz>

- Décompresser, installer
- Editer, créer le fichier **pb.mode**
- Dans une fenêtre "terminal", taper (entre [] = si c'est le cas)
`glpsol -model pb.mod [-data pb.dat -output pb.out]`

- **Installation sur Mac** :

- Si vous savez compiler sur Mac, suivre les instructions sur le site
<http://www.arnab-deka.com/posts/2010/02/installing-glpk-on-a-mac/>
- Si vous connaissez *darwinport*, installer glpk dernière version avec la commande "port" (ou l'outil *portauthority*)
- Si vous voulez juste l'exécutable (suffit pour ce BE), une solution "maline" est de récupérer Coliop3 qui contient glpk.
- ☞ Cette méthode fonctionne également pour les plateformes Linux et Windows. Son inconvénient est qu'on ne pourra pas utiliser *glpk* depuis un programme C/C++.

Installer GLPK (suite)

- **Passage par Coliop3 (toute plateformes) :**
 - Récupérer Coliop3 depuis le site
<http://www.coliop.org/download/Cmpl-1.7.1-osx.zip>
 - Décompresser et récupérer *glpsol* (l'exécutable pour ce BE) dans le répertoire Cmpl/Thirdparty/GLPK/glpsol
 - Vous pouvez ensuite récupérer les exemples et la documentation depuis n'importe quelle distribution de Glpk (par exemple Linux).
 - Utilisation :
 - Exécuter l'application "Terminal" (dans /Applications/Utilities/)
 - Placez vous dans le répertoire Cmpl/Thirdparty/GLPK
 - Avec un éditeur (*textedit*), créer ou éditer le fichier **pb.mod**
 - Dans la fenêtre "terminal", taper
`./glpsol -model pb.mod [-data pb.dat -output pb.out]` (entre [] = optionnel)
- Quelques documents déposés avec le sujet de ce BE.
- Pour Windows, il y a aussi "GLPK Lab".

Alternative : Installation de LPSOLVE

Si vous le souhaitez, pour récupérer LPSOLVE, aller sur la page :

<http://sourceforge.net/projects/lpsolve/files/lpsolve/5.5.2.0>

- Windows : récupérer le fichier exécutable `lp_solve_5.5.2.0_IDE_Setup.exe`
- Linux (32 bits) : `lp_solve_5.5.2.0_exe_ux32.tar.gz` (et `...u64...` pour 64 bits)
- Mac : `lp_solve_5.5.2.0_exe_osx32.tar.gz`
- Doc (il y a la doc disponible sur le Web) : `lp_solve_5.5.2.0_doc.tar.gz`
- Pour s'en servir sous Excel : Voir la même page.
- Pour OpenOffice : une extension existe chez OpenOffice.
 - ➔ Voir dans les fichiers déposés sur Pédagogie.
- Il est possible d'invoquer LPSOLVE depuis un programme :
 - ➔ récupérer sur la page ci-dessus le fichier `lp_solve_5.5.2.0_c.tar.gz`
- 📖 Lire le fichier (court) **`lp-sove-format.txt`** qui vous permet une prise en main rapide.

Plan (Chapitre 2)

- 1 Outils
 - Minizinc
- 2 Introduction aux éléments de syntaxe
 - Types
 - Ensemble
 - Type énuméré
 - Tableaux et chaînes
 - Les sorties
 - Opérateurs
 - Conversions
 - Itérations
 - Structure d'un programme
 - Type-inst
 - Contraintes
 - Array / Set en compréhension
 - Stratégies de résolution
- 3 Notre premier programme
 - Triangle
 - Coloration
 - Matrices dans MiniZinc
 - Les nombres réels
 - Réels : LP Simple
- 4 Notes sur Primal-Dual
- 5 Exemple moins triviaux
 - Giapetto
 - Modèle et Data pour Giapetto

Plan (Chapitre 2) (suite)

- Coloration plus générale
 - Optimisation du nombre de couleurs
 - Séparation Modèle-Data
- Zebre : Exemple de modélisation
 - Une solution Minizinc

6 Exercices à rendre

- Exemple Monnaie
- Exercice Paysans
- Exercice BIP (Trivial)
- Exercice : Gâteaux
- Exercice : Régression
- Exercice voyage
- Exercice coloration
- Exercice Fret
- Exercice Jobshop
- Exercice : Jésuites
- Exercice : Ateliers véhicules
- Exercice : Découpe papier
- Exercice Tournée (element)
- Exercice Bin Packing
- Tournoi
- PERT avec ressources (Cumulative)
- Sur les docks
- Infirmières
- Restaurant

Plan (Chapitre 2) (suite)

- Financement
- Voyageur
- Stockage

7 Contraintes Globales de Minizinc

8 Contraintes globales par famille

9 Annexes

- GLPK
- Installer LPSOLVE

10 Réels : Primal-Dual

- Giapetto
- Modèle et Data pour Giapetto

11 Exemple moins triviaux

- Coloration plus générale
- Optimisation du nombre de couleurs
- Séparation Modèle-Data
- PERT (BigM)
- Jobshop : définition de prédicat
- Zebre : Exemple de modélisation
 - Une solution Minizinc

12 Exercices en séance

- Exemple Monnaie
- Exercice Paysans

13 Exercices à rendre

- Exercice BIP (Trivial)

Plan (Chapitre 2) (suite)

- Exercice : Gâteaux
- Exercice : Régression
- Exercice voyage
- Exercice coloration
- Exercice Fret
- Exercice Jobshop
- Exercice : Jésuites
- Exercice : Ateliers véhicules
- Exercice : Découpe papier
- Exercice Tournée (element)
- Exercice Bin Packing
- Tournoi
- PERT avec ressources (Cumulative)
- Sur les docks
- Infirmières
- Restaurant
- Financement
- Voyageur
- Stockage

14 Contraintes Globales de Minizinc

15 Contraintes globales par famille

16 Annexes

- GLPK
- Installer LPSOLVE