Cours 7 Programmation par contraintes : Modélisation

Informatique Musicale
Master 2 - ATIAM

Pierre Talbot (talbot@ircam.fr)

UPMC/IRCAM

9 novembre 2017

Le menu

- Introduction
- Syntaxe de MiniZinc
- Modéliser des fonctions
- Les contraintes globales
- Conclusion

Programmation par contraintes

Paradigme de programmation

- Vous êtes habitué au paradigme impératif (C, C++, Java, . . .),
- peut-être objet (C++, Java, . . .),
- voir fonctionnel (OCaml, Haskell, Scala, . . .) ?

Aujourd'hui, on va voir le paradigme de la programmation par contraintes !

Programmation par contraintes

Paradigme déclaratif, surnommé "holy grail of computing" : on déclare notre problème et laisse l'ordinateur le résoudre pour nous.

Nurse Scheduling Problem

N infirmières travaillent M jours en rotation, trouver un planning tel que :

- Une rotation par jour et infirmière : jour, début nuit, fin nuit.
- Pas plus de deux nuits d'affilées, un jour pas précédé d'une nuit.
- ...

Interactive Soccer Queries

Suivant le système de score FIFA (0 défaite / 1 égalité / 3 victoire), répondre à des questions :

- Est-ce qu'une équipe peut encore devenir championne en théorie ?
- Est-ce qu'une équipe est sûr / à des chances de se qualifier ?
- ...

Problème d'harmonisation

Soit une série de N accords, trouver une permutation tel que le nombre de notes communes entre deux accords successifs soit maximisé.

Intuitions

Ces problèmes sont *combinatoires* et généralement *NP-complet*, très long voir impossible à résoudre sans optimisation.

Comment ça marche ?

On déclare un ensemble de variables et pose des contraintes/relations sur ces variables de tel sorte qu'on obtient une solution si toutes les contraintes sont vérifiées.

Premier exemple : Sudoku

Comment trouver les nombres manquants par un algorithme ? Difficile ?

Représentation mathématique

Soit une matrice M de taille 9×9 où $1 \leq M_{l,c} \leq 9$ et l est le numéro de la ligne et c celui de la colonne, on veut :

- Lignes différentes : $\forall l, c, c'.\ M_{l,c} \neq M_{l,c'}$ tel que $c \neq c'$.
- Colonnes différentes : $\forall l, l', c.\ M_{l,c} \neq M_{l',c}$ tel que $l \neq l'$.
- Sous-carré différents :
  $\forall l, c \in \{1, 4, 7\}.\forall sl, sc, sl', sc' \in \mathbb{Z}_3.\ M_{l+sl,c+sc} \neq M_{l+sl',c+sc'}$ tel que $sl \neq sl' \wedge sc \neq sc'$.

Représentation informatique

Modèle proche de la définition mathématique.
Pour résoudre : `solve satisfy;`

```
include "alldifferent.mzn";
int: N = 9;
array[1..N,1..N] of var 1..N: sudoku;
constraint forall(l in 1..N)
  (alldifferent([sudoku[l,c] | c in 1..N]));
constraint forall(c in 1..N)
  (alldifferent([sudoku[l,c] | l in 1..N]));
constraint forall(l, c in {1,4,7})
  (alldifferent([sudoku[l + sl, c + sc] | sl,sc in 0..2]));

solve satisfy;
output [show2d(sudoku)];
```

Input et Output

Une grille de Sudoku en entrée où les chiffres inconnus sont _.
Le système nous répond avec la grille pleine.

```
14 sudoku=[|
15 _, _, _, _, _, _, _, _, _|
16 _, 6, 8, 4, _, 1, _, 7, _|
17 _, _, _, _, 8, 5, _, 3, _|
18 _, 2, 6, 8, _, 9, _, 4, _|
19 _, _, 7, _, _, _, 9, _, _|
20 _, 5, _, 1, _, 6, 3, 2, _|
21 _, 4, _, 6, 1, _, _, _, _|
22 _, 3, _, 2, _, 7, 6, 9, _|
23 _, _, _, _, _, _, _, _, _|
24 |];
```

```
Compiling Sudoku.mzn
Running Sudoku.mzn
[| 5, 9, 3, 7, 6, 2, 8, 1, 4 |
   2, 6, 8, 4, 3, 1, 5, 7, 9 |
   7, 1, 4, 9, 8, 5, 2, 3, 6 |
   3, 2, 6, 8, 5, 9, 1, 4, 7 |
   1, 8, 7, 3, 2, 4, 9, 6, 5 |
   4, 5, 9, 1, 7, 6, 3, 2, 8 |
   9, 4, 2, 6, 1, 8, 7, 5, 3 |
   8, 3, 5, 2, 4, 7, 6, 9, 1 |
   6, 7, 1, 5, 9, 3, 4, 8, 2 |]
----------
Finished in 8msec
```

Démo live.

Le menu

- Introduction
- Syntaxe de MiniZinc
- Modéliser des fonctions
- Les contraintes globales
- Conclusion

Quels outils pour résoudre un problème de contraintes ?

- *Librairies* : GeCode (C++), Choco (Java), ...
- *Langages* : Prolog, **MiniZinc**, ...

Pourquoi MiniZinc ?

- Facilité : Syntaxe proche du raisonnement mathématique.
- Modulaire : Compile le code vers différents solveurs (GeCode, Choco, ...).
- Encore plus modulaire : Compile vers différents "sous-paradigmes" (CP, ILP, MIP, ...). Avec le même code !

Déclarer des variables

```
int : n = 9; % Parametre
var 1..9 : y; % Variable
```

Deux types de variables

- *Paramètre* : La variable `n` est un paramètre qui est fixé avant l'exécution et ne peut prendre qu'une seule valeur. Dans le problème du Sudoku, c'est la taille de la matrice.
- *Variables de décision* : La variable `y` prendra une valeur entre `1` et `9` à la fin de l'exécution. Dans le Sudoku, c'est par exemple une case de la matrice.

Types des variables

`int`, `bool`, `float`, `string`, `set` and `array`.

Ajouter des contraintes

Une contrainte est une relation bidirectionnelle sur les variables. Exemple :

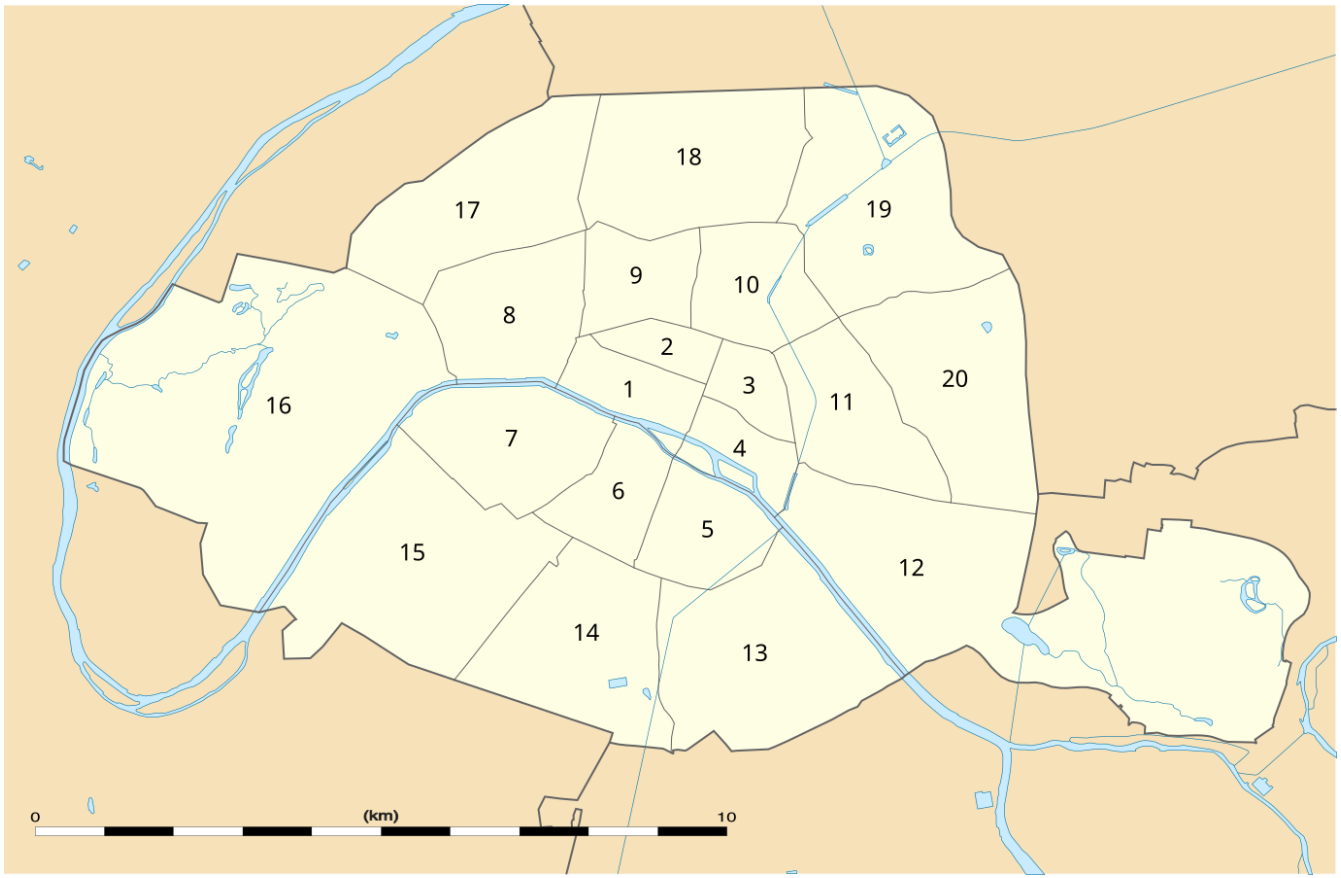
```
var 1..10 : x;
var 1..10 : y;
int : n = 4;
constraint x = y + n;
```

Si x change, ça impact y qui change pour respecter l'égalité, et inversement.

- Contraintes arithmétiques : `x < y` (les classiques : `<,<=,>,>=,=`)
- Expressions arithmétiques : `y - z = x + 3` (`-,+,*,/,mod,div`).
- Contraintes booléennes : `x != y \/ y = 0` (`\/,/\,->,<->,not`).

Exemple : Colorier une carte

Selon la carte suivante, trouver un coloriage à 3 couleurs des arrondissements 3 à 5 et 11,12 tel que 2 arrondissements adjacents n'ont pas la même couleur.
Squelette sur la page du cours : hyc.io/teaching/im.html



Solution : Colorier une carte

```
int : nc = 3;

var 1..nc : arr3;
var 1..nc : arr4;
var 1..nc : arr5;
var 1..nc : arr11;
var 1..nc : arr12;

constraint arr3 != arr4;
constraint arr3 != arr11;
constraint arr4 != arr11;
constraint arr4 != arr12;
constraint arr4 != arr5;
constraint arr11 != arr12;

solve satisfy;
output ["arr3 = \(arr3)\t arr4 = \(arr4)\t arr5 = \(arr5)\n",
        "arr11 = \(arr11)\t arr12 = \(arr12)"];
```

Exercice : Puzzle d'Abbot

On distribue 100 caisses de maïs à 100 personnes tel que :

- Chaque homme reçoit 3 caisses, chaque femme 2 et chaque enfant une moitié.
- Il y a 5 fois plus de femmes que d'hommes.
- Trouver une répartition hommes, femmes et enfants.

Exercice : Qui a triché ?

Trois élèves sont interrogés pour savoir si quelqu'un a triché.

- A : Il y a un tricheur.
- B : Il y a deux tricheurs.
- C : Il y a trois tricheurs.

Les élèves qui trichent mentent toujours et ceux qui ne trichent pas disent la vérité. Qui triche(nt) ?

Plus sur MiniZinc

Ligne de commande :

- `minizinc paris-color.mzn` : Trouve la première solution.
- `minizinc -a paris-color.mzn` : Trouve toutes les solutions.

Dans l'IDE, cocher la case "Print all solutions" dans l'onglet configuration.

Architecture MiniZinc

- Modèle (".mzn") + Données (".dzn") donne un FlatZinc (".fzn") qui est donné à un solveur pour la résolution.
- Les ".dzn" fixent les paramètres d'un modèle (exemple : `nc = 2;`).
- `minizinc paris-color.mzn paris-color.dzn`
- `minizinc paris-color.mzn -D"nc = 2;"`

Problème d'optimisation

- On veut trouver une valeur maximum ou minimum parmi toutes les solutions suivant un certain critère.
- On remplace `solve satisfy;` par :
  1. `solve maximize x+y;` pour maximiser l'expression $x + y$.
  2. `solve minimize a;` pour minimiser la variable a.
- Très fréquent dans la "vraie vie", on veut souvent minimisez le coût, maximiser la productivité, ...

Colorier une carte – Revisité

On veut minimiser le coût d'impression :

- On a trois couleurs : blanc, noir et rouge.
- L'impression d'un pays en blanc coûte 1, en noir 2 et en rouge 3.
- Quel est le coût minimum ?

Solution : Colorier une carte – Revisité

- Astuce : le numéro de la couleur est aussi son côut.
- Notez qu'on donne une borne supérieure à tarif.
- Pour la performance il est important d'essayer de *toujours donner une borne supérieure aux variables*.

```
var 1..nc*5 : tarif = arr3 + arr4 + arr5 + arr11 + arr12;

solve minimize tarif;
```

Que faire si les tarifs sont : blanc (1), noir (1) et rouge (4) ?

Le menu

- Introduction
- Syntaxe de MiniZinc
- Modéliser des fonctions
- Les contraintes globales
- Conclusion

Problème d'affectation

Pour des objets dans un domaine, trouver les objets correspondants d'un codomaine.

Exemples

- Soit le domaine des arrondissements $\{1, .., 5\}$, donner une correspondance couleur dans l'ensemble $\{1, .., 3\}$.
- Soit le domaine des couleurs $\{1, .., 3\}$, donner une correspondance prix dans l'ensemble $\{1, 4\}$.

Les ensembles

Rappel

- Les paramètres (`int: x = 1;`) sont instanciés dès le début.
- Les variables de décision (`var 1..5: vx;`) s'instancient au cours de l'exécution : 1..5 représente l'ensemble des valeurs *possibles*.

Ensemble

- `set of int: ARR = 1..5;` représente l'ensemble d'entier $\{1, .., 5\}$.
- Quelle différence avec `var 1..5: arr;` ?

Les ensembles

Rappel

- Les paramètres (`int: x = 1;`) sont instanciés dès le début.
- Les variables de décision (`var 1..5: vx;`) s'instancient au cours de l'exécution : 1..5 représente l'ensemble des valeurs *possibles*.

Ensemble

- `set of int: ARR = 1..5;` représente l'ensemble d'entier $\{1, .., 5\}$.
- Quelle différence avec `var 1..5: arr;` ? `ARR` restera un ensemble mais `arr` deviendra finalement un `int`.

Les tableaux

- Les indices d'un tableau sont représentés par un ensemble d'entier.
- La correspondance entre DOM et CODOM peut être statique (`price`).

DOM arrondissements $\{1, .., 5\}$ vers CODOM couleurs $\{1, .., 3\}$.

```
int : arr3 = 1 ; int : arr4 = 2 ;
set of int : ARR = 1..5 ;
set of int : COLOR = 1..3 ;
set of int : PRICE = {1, 4} ;
array[ARR] of var COLOR : color ;
array[COLOR] of PRICE : price = [1, 1, 4] ;

constraint color[arr3] != color[arr4] ;
...
```

Comment utiliser ces tableaux dans des contraintes ?

Générateurs

Qui dit tableaux... dit boucles. Deux catégories d'opérations :

- *Fold* : Parcours un tableau pour construire un unique objet (exemple : la somme d'un tableau).

```
array[1..n] of var 1..c : cost;
var 1..n*c : total = sum(cost);
```

- *Map* : Parcours un tableau pour construire un autre tableau (exemple : multiplier tous les éléments par deux).

```
array[1..n] of var 1..c*2 : double_cost =
  [cost[i] * 2 | i in 1..n];
```

Générateur Map/Filter (compréhension de liste)

Soit une liste d'élément, pour chaque élément respectant certains critères (filtrage) on leur applique une opération (map) et retourne la nouvelle liste :

```
% neighbors[i,j] correspond au score que i donne a j.
array[1..n, 1..n] of 0..p : neighbors;
% On calcule le score de chacun, on utilise une double comprehension.
array[1..n] of 0..p*n : score =
  [sum([neighbors[i,j] | j in 1..n where i != j])
   | i in 1..n];
```

Générateur Fold

Soit une liste d'élément, retourne un résultat. Reprenant l'exemple précédent, on peut calculer le score globale du groupe de personne :

```
var 0..p*n*n : global_score =
  sum([neighbors[i,j] | i,j in 1..n where i != j]);
```

Sucre syntaxique

Au lieu de faire `sum([e | g])` on peut écrire `sum(g)(e)` ce qui est plus clair :

```
var 0..p*n*n : global_score =
  sum(i,j in 1..n where i != j)
     (neighbors[i,j]);
```

Générateur Fold

Similaire à *sum*(*array*) on a *product*(*array*).
Pour générer une conjonction de contraintes, on peut utiliser *forall*(*array*) :

```
int : n = 3;
constraint forall(i,j in 1..n where i < j)
                    (t[i] != t[j]);
```

C'est transformé **avant** l'exécution en une conjonction de contraintes :

```
constraint t[1] != t[2] /\ t[1] != t[3] /\ t[2] != t[3];
```

Colorier une carte – Avec tableau

À partir de l'exemple ci-dessous, ré-écrire l'optimisation du coloriage d'une carte avec les tableaux.

```
int : arr3 = 1 ; int : arr4 = 2 ;
set of int : ARR = 1..5 ;
set of int : COLOR = 1..3 ;
set of int : PRICE = {1, 4} ;
array[ARR] of var COLOR : color ;
array[COLOR] of PRICE : price = [1, 1, 4] ;

constraint color[arr3] != color[arr4] ;
...
```

Le menu

- Introduction
- Syntaxe de MiniZinc
- Modéliser des fonctions
- Les contraintes globales
- Conclusion

Sous-structure d'un problème

Il y a beaucoup de problème qui montre des sous-structures identiques, par exemple : "tous les éléments de ce tableau doivent être différents".

Contraintes globales

- Contraintes n-aires capturant un sous-problème particulier.
- Abstraction réutilisable dans plusieurs problèmes.
- Leur implémentation est généralement plus efficace que celle de leur décomposition.

Contrainte alldifferent

Le prédicat `alldifferent(array)` est fourni par la bibliothèque standard de MiniZinc. On doit inclure les contraintes globales utilisées.

```
include " alldifferent .mzn";
int : N = 9;
array[1..N,1..N] of var 1..N : sudoku;
constraint forall(l in 1..N)( alldifferent ([sudoku[l,c] | c in 1..N]));
```

Note : On peut utiliser `include "globals.mzn"` pour inclure toutes les contraintes globales disponibles.

Problèmes des N-reines

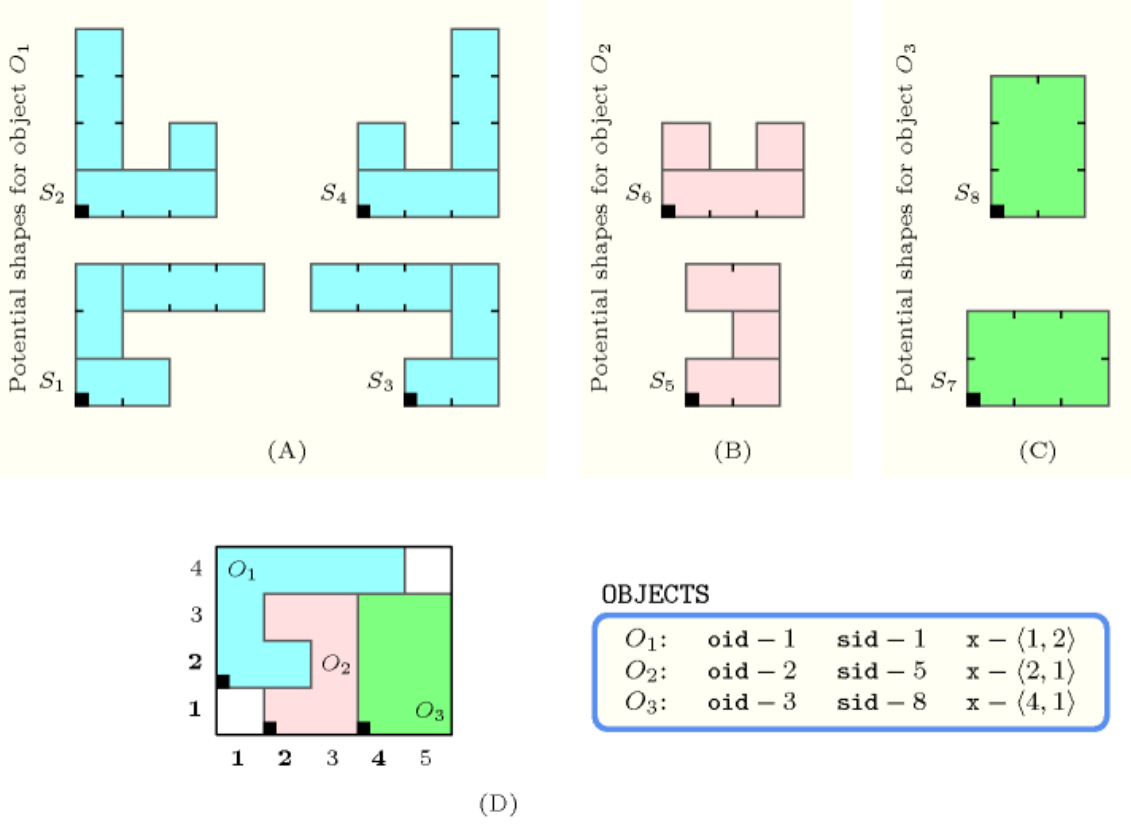
Soit un échiquier de taille N * N, placer sur chaque ligne une reine tel que aucune reine ne puisse s'attaquer (ligne, colonne, diagonale).
Tester avec "N = 35", est-ce que les contraintes globales aident à réduire le temps d'exécution ?

Packing Problem

- Répartition d'un chargement de camion en équilibrant le poids.
- Plan optimal d'un appartement pour maximiser le nombre de chambres (d'une taille min).
- Plan de circuit intégré : minimiser le coût en plaçant les blocs suivant certaines contraintes.

Geost

Ces problèmes ont une sous-structure commune capturée par la contrainte globale `Geost(k, Objects, SBoxes)`.



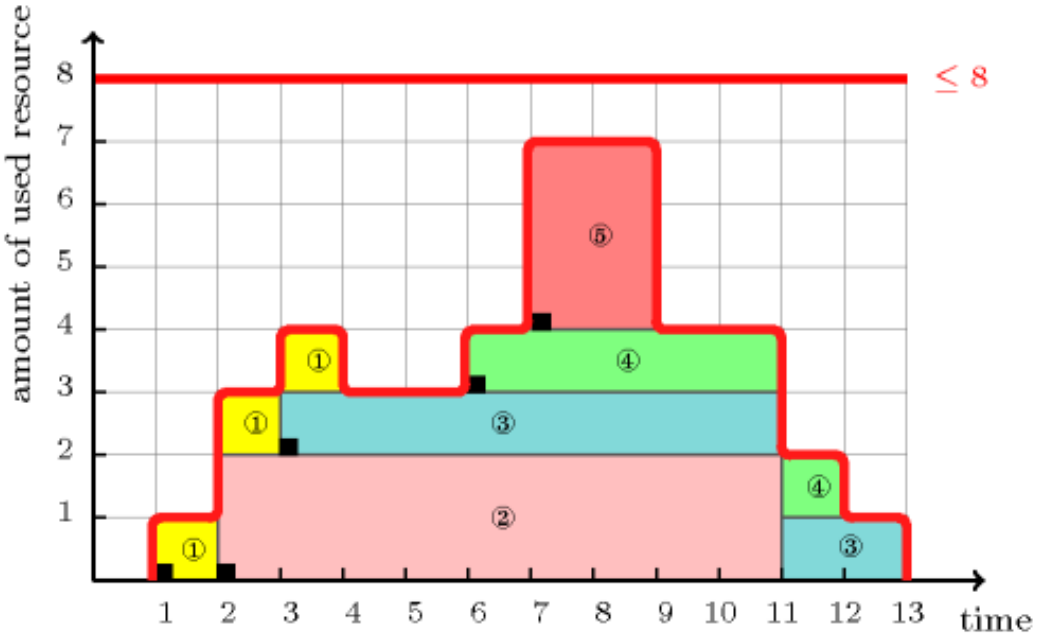
Ordonnancement de tâches sous ressources

- Établir un planning de cours.
- Maximiser la productivité de machine dans une usine.
- ...

Cumulative

```
predicate cumulative(array [int] of var int : s,
                     array [int] of var int : d,
                     array [int] of var int : r,
                     var int : b)
```

Soit un ensemble de tâches, une tâche i commence au temps $s[i]$ pour une durée de $d[i]$ et utilise $r[i]$ ressources. Le nombre de ressources maximales par instant est de b (sur le schéma $b = 8$).



Catalogue de contraintes globales

Il existe de nombreuses contraintes globales ($>$400) dont une partie est référencée sur ce site : `http://sofdem.github.io`.

- Pas de panique, c'est surtout des variantes, il existe beaucoup moins de classe de problèmes.
- On a vu certaines classes et il est possible de relier des problèmes musicaux à celles-ci (thèse Truchet).

Le menu

- Introduction
- Syntaxe de MiniZinc
- Modéliser des fonctions
- Les contraintes globales
- Conclusion

Conclusion

La programmation par contraintes permet principalement de résoudre des problèmes combinatoires et NP-complet.

- On déclare et le système le résout automatiquement.
- On se concentre sur le problème plutôt que sa méthode de résolution.
- Il existe de nombreux sous-paradigmes plus ou moins efficaces pour certaines classes de problèmes (local search, MIP, SMT...).

Ceci n'est qu'une introduction...

- Modélisation de graphes, table associative, ...
- Symmetry breaking
- Les stratégies d'exploration : IDDFS, LDS, ...
- Parallélisation (EPS, ...)

En savoir plus

Je vous conseille les cours et articles suivants :

- Coursera – Modeling Discrete Optimization (P. Stuckey, J. Lee) : Focus sur la modélisation par programmation par contraintes (on a vu des notions de la 1er et 2eme semaine de ce cours).
- Coursera – Discrete Optimization (P. Van Hentenryck) : Cours plus général sur les différents paradigmes (CP, MIP, local search).
- Guido Tack. Constraint Propagation Models, Techniques, Implementation
- Gérard Assayag et Charlotte Truchet. Constraint Programming in Music
- Pour d'autres articles de recherche ou renseignements, me contacter ;-)