Recherche Opérationnelle
& Programmation avec Contraintes

Chapitre 1 - Optimisation Combinatoire

Alexandre Saidi
École Centrale de Lyon
Département Mathématiques-Informatique
LIRIS - CNRS

Novembre 2017

Introduction & Plan

- ✓ Cadre général : problèmes d'Optimisation Combinatoires
  - Recherche Opérationnelle (OR) et *Prog. Mathématique / Linéaire*
  - Techniques de Satisfaction / Programmation avec Contraintes (CSP / CP)
  - Comparaisons et Intégration des deux "écoles"

- ✓ Deux grands Chapitres :
  - Modélisation de problèmes
  - Résolution de problèmes :
    - → Faisabilité (eg. Algos de filtrage de contraintes globales)
    - → Optimalité (des solutions) via :
      - ➙ Branch and Bound (CP, OR)
      - ➙ Cutting Plans & Branch and Cut (OR, CP)
      - ➙ Génération de colonnes (OR, CSP), ...
    - → Recherche (dans un espace d'états)
      - ➙ Propagation de Contraintes (+ Retours arrières en CP / CLP)

- ✓ 4 crs, Multiples Exemples, Un BE, ... & MiniZinc

Cadre et contexte

- Recherche Opérationnelle (OR) :

  Une branche des **Mathématiques Appliquées** qui utilise

  - *la Modélisation Mathématique*, les *Probabilités et Statistiques*,
  - les *Méthodes Algorithmiques*, la *Théorie de Graphes,*
  - *la théorie des jeux et des Files d'attentes*,
  - la *Simulation* et l'*Analyse décisionnelle*

  pour obtenir ou approcher une solution optimale aux problèmes complexes.

- OR traite des techniques pratiques d'optimisation :
  modéliser, définir et trouver le minimum ou le maximum d'une fonction objective portant sur (p. ex.) des :

  - Rendements (en agriculture), Performances d'une ligne d'assemblage,
  - Profits, Bandes passantes, Pertes / Gains, Temps d'attente,
  - Risques et Fiabilités, Durée de garanti, etc ...

Cadre et contexte ...suite

- OR est un outil d'*aide au management* en utilisant un processus scientifique
  - ➛ Le "management" apporte des synonymes de OR (dans la littérature) : "science de management", "science de décision" , ...
- Nature des problèmes visés :
  - Problèmes NP-Complets/NP-difficiles
  - Avec la Minimisation/Maximisation d'une fonction objective dont les variables sont soumises aux :
    - Contraintes arithmétiques
    - Contraintes Symboliques
    - Contraintes logiques (e.g. disjonction / implication sur des expressions de variables)

☞ Dans certains cas, on peut ne pas avoir une fonction objective (e.g. Sudoku)

  - Il y a également des contraintes dites globales (vs. locales)
    - ➛ $X > Y$ est plutôt locale
    - ➛ *increasing* est globale (*MinZinc*)

Cadre et contexte ...suite

- Domaines d'applications :
  - Optimisation d'"affectation" sous contraintes,
  - Allocation des ressources,
  - Scheduling,
  - Planning (Emploi du temps, ...),
  - Routage, Séquencement (chaîne de production),
  - Design, Configuration,
  - Bin-Packing,
  - Flux
  - Et beaucoup d'autres

Remarques :

En OR, la "Programmation Mathématique" couvre :

- La programmation Linéaire, en nombre entiers, bool, réels, etc. voire chaînes, listes, ensembles, symboliques
- La programmation non linéaire en fait également partie

Propos / objectifs

- **Modèle conceptuel** :
  - Créer un modèle (*conceptuel*) qui fait **abstraction** d'un problème **réel**
  - Créer une formulation avec des contraintes qui **modélise** cette abstraction (et donc le problème initial)
  - .. Et la formulation a une chance d'être solvable (résoluble)

- Exemple (simple) : *affecter une somme totale $S > k$ à k individus (x_i) tel que $x_i > 0$ et chacun reçoive une valeur différente*

- **Modèle mathématique** :

$$S = \sum_{i=1}^{k} x_i$$
$$\wedge \quad x_i > 0, x_i \in \mathbb{N} \quad integer(x_i)$$
$$\wedge \quad \forall i, j \in 1..k, \quad i \neq j \implies x_i \neq x_j, \qquad \text{ou } alldifferent(x_1..x_k)$$

➙ Le modèle mathématique n'est (en théorie) pas restreint par l'environnement

Propos / objectifs ...suite

- **Modèle d'implantation** :

1. Programmation linéaire en nombre entiers (PLNE) :
   Créer un programme (un *système*) LP qui résout notre formulation

   ➛ Il faut énumérer toutes les k variables, et **par extension** (en détails)

   $$\text{poser : } S = \sum_{i=1}^{k} x_i,\ \forall i \neq j \in 1..k, x_i \neq x_j, \text{ préciser } integer(x_i)$$

   ☞ La contrainte $\neq$ est non linéaire, elle n'existe pas en LP !

2. Programmation avec contraintes (en nombre entiers) en CP/CLP :

   ➛ $X = liste(k); \forall x \in X : x > 0; S = sum(X); alldifferent(X)$ ...

   ☞ Par son niveau d'abstraction, la CSP/CP/CLP est un outil de modélisation.

Question de test / trace / déverminage :

➛ Au moment des tests individuels, la valeur de x_i n'est pas connue !

Propos / objectifs ...suite

Un autre exemple (partiel) :

- **Modèle conceptuel** :
  - Besoin d'imposer que *seules 2 valeurs différentes* soient présentes dans une séquence de variables (e.g. dans un vecteur **V**)
- **Modèle mathématique** :
  $\exists i, j \ i \neq j \in 1..|V|, V_i \neq V_j \wedge \forall k, l \in 1..|V|, \ k \neq l \neq i \neq j, \ V_k = V_l$
- **Modèle d'implantation** :
  - Sol 1 : trouver *min* et *max* de V, imposer $min \neq max$, compter le nombre d'occurrences de chacun et imposer que ce nombre $= 2$
  - Sol 2 : Transformer **V** et un ensemble (*set*) **S** puis calculer *card(S)*;
  - Sol 3 : contrainte (CP) *nvalues(V,2)* (*nvalues(2,V)* en MiniZinc)
  - Sol 4 : ...
- **Exercice** : trouver ces 3 modèles pour que :
  parmi les valeurs de V, *seules 2* puissent identiques / différentes.

Propos / objectifs ...suite

- La démarche décrite est souvent itérative où l'on applique :
  - Des techniques différentes (sur un graphe, une matrice, un tableau, ...)
  - Des modèles différents (LP, CSP, CLP, ...)
  - Des Heuristiques différentes (propres au problème donné)

- Le modèle **conceptuelle** vient avec la pratique (pas de pilule !)

- Le modèle **implantation** : différentes approches possibles :
  - Langage impératifs (objet) avec des librairies ad-hoc
  - Langages CSP (CLP, sans parler des *vintages* !)
  - Langage de modélisation mathématiques (e.g. OPL, ZIMPL)
  - Des langages spécifiques (produits "maison", avec des techniques ad-hoc codées)

- Ce qui **varie** :
  - Le niveau d'abstraction : ± proche de la machine ou du modèle conceptuel ?
  - Leur pouvoir d'expression (e.g. *alldifferent()*)
  - Leur complexité d'utilisation / d'exécution

Caractéristiques CSP/CP & OR

- Caractéristiques de la **CP** (Programmation avec des Contraintes)
  - Modélisation déclarative
  - La logique du 1er ordre disponible
  - Propagation des Contraintes
  - Recherche (dans un espace d'états) selon différentes algos / stratégies

- Caractéristiques de **OR** (Recherche Opérationnelle, autres que *Simplex*) :
  ➛ Techniques qui interviennent dans la résolution (par étapes) :
  - *Branch and bound* (relaxation, e.g. en programmation Linéaire)
  - *Branch and Cut* (cutting planes)
  - *Génération de colonnes*

- Remarques : méthodes "Complètes"/"Partielles"
  - Possibilité d'obtenir une / toutes les solutions, *Contraintes-réponses*, ...
    ➛ *Contrainte-réponse* et propagation de contraintes
  - Idée d'Heuristique (ou de *méta-heuristique*)

Pour 18-19

• Voir l'ex Time Table que j'ai mis dans le BE. On peut s'en servir ici comme ex de modélisation ? Avec sa sol en LP, ...Et dans d'autres langages (voir livre "Hybrid Metaheuristics")

Un exemple

• Un fabricant de jouets cherche à savoir combien de bicyclettes (B) ou de tricycles (T) doit-il fabriquer pendant une semaine de 40 heures sachant que :

- L'usine peut produire par heure 200 bicyclettes ou bien 140 tricycles
- Le profit sur une bicyclette est de 25 et de 30 pour un tricycle
- On ne peut pas vendre plus de 6000 bicyclettes et 4000 tricycles par semaine.

• **Formulation LP :**

$$
\begin{aligned}
max \quad & Z = 25B + 30T \\
s.t. \quad & \frac{B}{200} + \frac{T}{140} \leq 40 \\
& 0 \leq B \leq 6000, 0 \leq T \leq 4000 \\
& B, T \in \mathbb{N} \qquad (B, T \ entier)
\end{aligned}
$$

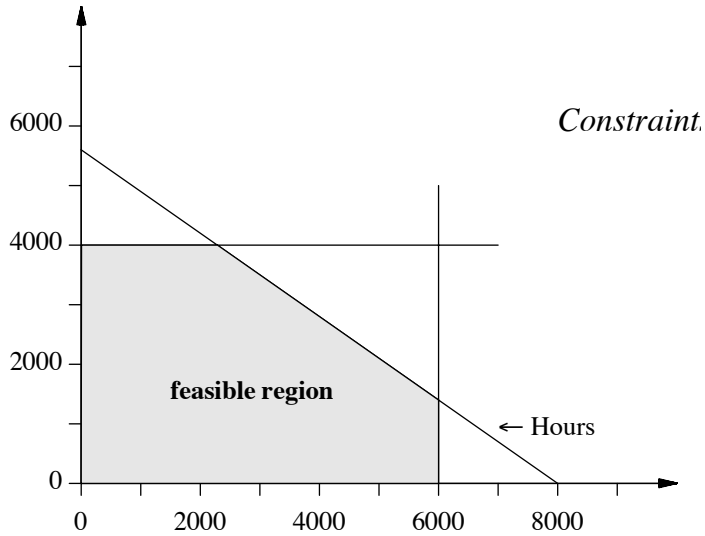
➙ Traduction P. ex. en *LpSolve* (un outil LP) est quasi directe.

Un exemple ...suite

Interprétation géométrique :

Rappel du modèle :

$$
\begin{array}{ll}
max & Z = 25B + 30T \\
s.t. & \frac{B}{200} + \frac{T}{140} \leq 40 \\
 & 0 \leq B \leq 6000, 0 \leq T \leq 4000 \\
 & B, T \in \mathbb{N} \qquad (B, T \ entier)
\end{array}
$$



➡ Les 2 axes : les deux produits

- Solution optimale (sommet à droite) : *B=6000, T=1400, Z=192000.*

Un exemple ...suite

- **L'approche par un langage impératif :**

Rappel du modèle :

$$
\begin{aligned}
max \quad & Z = 25B + 30T \\
s.t. \quad & \frac{B}{200} + \frac{T}{140} \leq 40 \\
& 0 \leq B \leq 6000, 0 \leq T \leq 4000 \\
& B, T \in \mathbb{N}
\end{aligned}
$$

Un langage impératif

Mélangera P. ex. les structures et les instructions de C++ (ou Java) avec des appels à une bibliothèque de fonctions spécifique.

Par exemple : `CPXLPptr lp=CPXcreateprob(...);`

pour créer une instance permettant d'appeler d'autres fonctions (sur *lp*).

Un exemple ...suite

- **L'approche objet** (plus élégante) :

Rappel du modèle :

$$
\begin{aligned}
max \quad & Z = 25B + 30T \\
s.t. \quad & \frac{B}{200} + \frac{T}{140} \leq 40 \\
& 0 \leq B \leq 6000, 0 \leq T \leq 4000
\end{aligned}
$$

L'approche objet

Par exemple, dans un C++ hypothétique proche de *G12* :

```
#include <intsolver.h>
CPLEXIntSolver s();
CpInt B(s,0,6000), T(s,0,4000);
s.AddConstraint((1.0/200.0)*B + (1.0/140.0)*T ≤ 40.0);
s.MaxObjective(25.0*B+30.0*T);
s.Solve();
cout << "B= " <<B.val() << "T=" << T.val();
```

Un exemple ...suite

- **Outil dédié : AMPL**

Rappel du modèle :

$$
\begin{aligned}
max \quad & Z = 25B + 30T \\
s.t. \quad & \frac{B}{200} + \frac{T}{140} \leq 40 \\
& 0 \leq B \leq 6000, 0 \leq T \leq 4000
\end{aligned}
$$

Avec AMPL :

```
var B; var T;
maximize profit: 25*B+30*T;
subject to: (1/200)*B+(1/140)*T <= 40;
subject to: 0 <= B <= 6000;
subject to: 0 <= T <= 4000;
solve;
```

☞ Le choix du solveur (ici *Integer*) détermine le domaine de B et de T.

➛ AMPL est juste un *wrapper* (qui traduit le modèle pour un solveur fourni)

Un exemple ...suite

- **CLP (Eclipse/ PrologIII/ PrologIV/ CLPR/ ...)**

Rappel du modèle :

$$
\begin{array}{rr}
max & Z = 25B + 30T \\
s.t. & \frac{B}{200} + \frac{T}{140} \leq 40 \\
 & 0 \leq B \leq 6000, 0 \leq T \leq 4000
\end{array}
$$

Solution CLP (peuvent faire appel à un solveur simplex ou pas

```
résoudre(B,T) ←
  0 <= B <= 6000,
  0 <= T <= 4000,
  (1/200)*B+(1/140)*T <= 40,
  maxof(25*B+30*T).
```

➙ Domaine par défaut : dans $\mathbb{Z}$ (ici $\mathbb{N}$ à cause des intervalles).

Un exemple ...suite

Rappel du modèle :

$$
\begin{aligned}
max \quad & Z = 25B + 30T \\
s.t. \quad & \frac{B}{200} + \frac{T}{140} \leq 40 \\
& 0 \leq B \leq 6000, 0 \leq T \leq 4000
\end{aligned}
$$

- **Solution NumberJack avec Python** :

```
def Monmodel :
  B = Variable(0..6000)
  T = Variable(0..4000)
  model = Model(
        Minimise(25*B+30*T),
        [(1/200)*B + (1/140) * T <= 40]
                )
  solver=Monmodel.model.load('Mistral') # un solveur
  solver.solve()
```

➙ Même solution..

Un exemple ...suite

Rappel du modèle :

$$
\begin{aligned}
max \quad & Z = 25B + 30T \\
s.t. \quad & \frac{B}{200} + \frac{T}{140} \leq 40 \\
& 0 \leq B \leq 6000, 0 \leq T \leq 4000
\end{aligned}
$$

- **Solution MiniZinc** :

```
var 0..6000: B;
var 0..4000: T;
constraint (1.0/200.0)*B+(1.0/140.0)*T <= 40.0;
solve maximize 25.0*B + 30.0*T;
output ["B = ", show(B), "T = ", show(T), "\n"];
```

➙ N.B. activer l'option "-b mip" (ou "G12MIP" dans l'onglet "Configuration")

- Etc.....

Exemples d'environnements

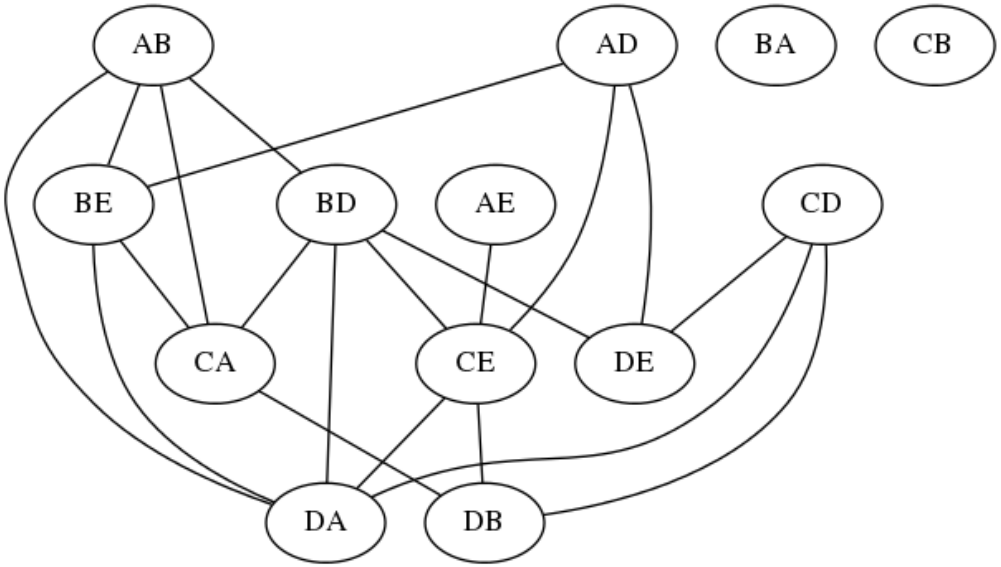
Quelques environnements (non exhaustive):

<table>
<tr><th></th><th>CLP</th><th>C++ toolkit</th><th colspan="2">Modelling language</th></tr>
<tr><td>LP/ILP</td><td rowspan="3">ECLiPSe</td><td>Mistral</td><td rowspan="2">OPL</td><td rowspan="2">MiniZinc</td></tr>
<tr><td>BT search + propagation</td><td>ILOG Solver</td></tr>
<tr><td>Local search</td><td>Localizer ++</td><td>Comet</td><td></td></tr>
</table>

Ex. de Modélisation : problème de coloration

- Il existe plusieurs modélisations / formulations OR / CSP de ce problème
  - Basique (Graphe, noeuds et arête)
  - A base d'ensembles indépendants
  - Comme un problème de *Scheduling / Affectation* (matrices)
  - ...
- Fonction d'optimisation de la coloration :
  - Minimiser le nombre de couleurs utilisées.
  - Imposer que les couleurs soient utilisées dans l'ordre $1..|K|$, ...
- Applications : feu de circulation, organisation d'examens, ...
  - Graphe de contraintes vs. graphe de compatibilité
- Idée d'implantation (langages impératifs): algorithme Gloutonne (expliquer !)
- Idée : Heuristique

Application : Feux de circulation

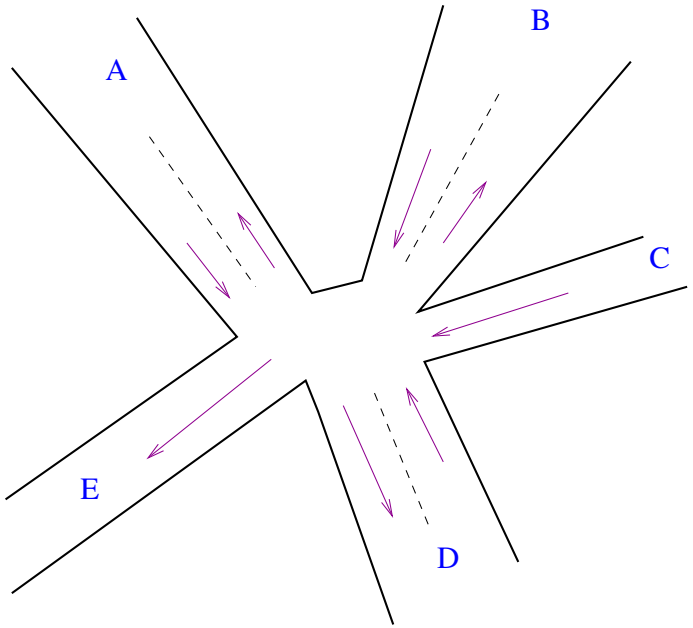


Modélisation :

- Les variables : tous les noeuds
  ➙ Ex. $AB, BE, ...,$
- Domaine des variables : $1..Nb\_colors > 3$
- Si deux noeuds x, y reliés dans le graphe
  Alors poser la contrainte $x \neq y$

Une solution possible :

$Couleur_1 : AB, AD, BA, CB, AE, CD$
$Couleur_2 : BE, BD, DB$
$Couleur_3 : CA, CE, DE$
$Couleur_4 : DA$



Les circulations autorisées :
➙ depuis A : $AB - AD - AE$
➙ depuis B : $BA - BD - BE$
➙ depuis C : $CA - CB - CD - CE$
➙ depuis D : $DA - DB - DE$

Application : Feux de circulation ...suite

Implantation CLP/CSP

```
colorer(Noeuds,K) :-          % utiliser K couleurs (minimisation non traitée)
    Noeuds = [AB,AD,BA,CB,BE,BD,AE,CD,CA,CE,DE,DA,DB],
    Noeuds :: 1..K,
    dif(AB, BD), dif(AB, BE), dif(AB, CA), dif(AB, DA),
    dif(AD, BE), dif(AD, CE), dif(AD, DE),
    dif(AE, CE),
    dif(BD, CA), dif(BD, CE), dif(BD, DA), dif(BD, DE),
    dif(BE, CA), dif(BE, DA),
    dif(CA, DB),
    dif(CD, DA), dif(CD, DB), dif(CD, DE),
    dif(CE, DA), dif(CE, DB),
    labeling(Noeuds).
```

Une solution :

```
?-colorer(L,4).
        [AB,AD,BA,CB,BE,BD,AE,CD,CA,CE,DE,DA,DB],
        [1 ,1 ,1 ,1 , 2, 2, 1, 1, 3, 3, 3, 4, 2 ]
```

- Les noeuds qui ont la même couleur sont compatibles (feu vert simultané)
- Ou jaune clignotant simultané.

Coloration : Modélisation IP/BIP

Ex. 2 avec un graphe plus simple :

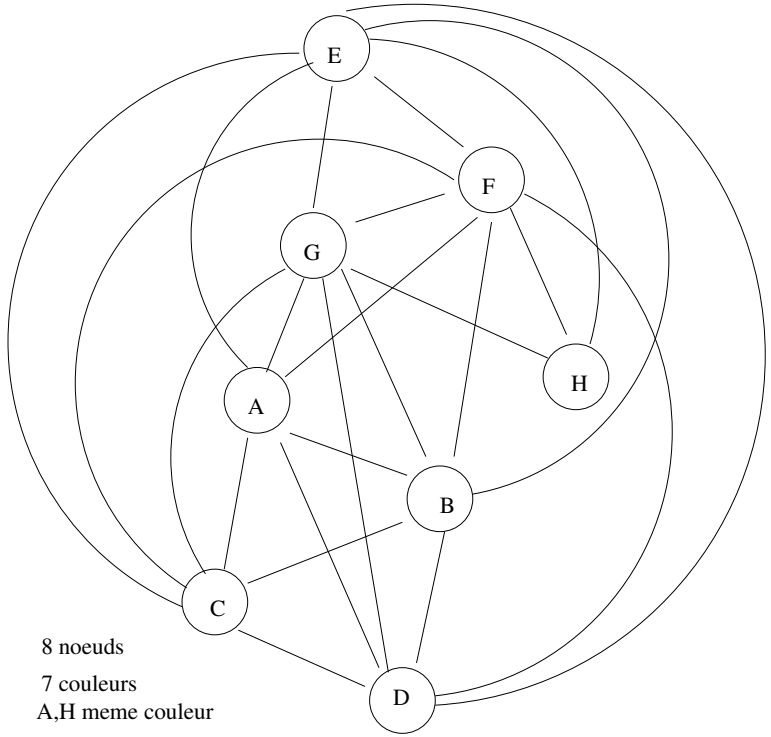
- graphe $G = (V, E)$
- ensemble de couleurs K (ici $K = 7$)

- **Une formulation IP (BIP)** :

  Soit X[V,C] la matrice (de bool) ci-dessous

- $X_{v,c} = 1$ si le noeud v est coloré par la couleur c, 0 sinon
- $\forall v \in V, \sum_{c=1}^{k} X_{v,c} = 1$ pour l les lignes
- $X_{u,c} + X_{v,c} \leq 1 \quad \forall c \in K, \forall$ *arete* $\{u, v\} \in E$

	c1	c2	c3	c4	c5	c6	c7
A	V	F	F	F	F	F	F
B	F	V	F	F	F	F	F
C	F	F	V	F	F	F	F
D	F	F	F	V	F	F	F
E	F	F	F	F	V	F	F
F	F	F	F	F	F	V	F
G	F	F	F	F	F	F	V
H	V	F	F	F	F	F	F



- **Rappel formulation CP**

color(V, E) :-
V=[A,B,C,D,E,F,G,H] :: 1..K,
$\forall X \in V, Y \in V, X \neq Y :$
$(X, Y) \in E \implies dif(X, Y),$
labeling(V).

Implantation MiniZinc

Une solution Minizinc sans optimisation du nombre de couleurs :

```
% nombre de noeuds
int: n = 10;

% Data : ensemble des arêtes (Deux noeuds par arête)
array[1..n] of set of int: neighbors;

% variables de décision
array[1..n] of var 1..4: x;

solve satisfy;

% Deux voisins n'auront pas la même couleur
constraint
    forall(i in 1..n, j in neighbors[i]) (
        x[i] != j
    );

% Les sorties
output [ show(x)];
```

Le graphe est exprimé a part (dans une section *Data*) $\cdots \rightsquigarrow$

Implantation MiniZinc ...suite

```
% Data : les données du problèmes
%
neighbors =[
    {},
    {1},
    {1,2},
    {1,3},
    {1,4},
    {1,5},
    {1,6},
    {1,7},
    {1,8},
    {1,9},
    {1,10},
    {2,3},
    .....
];


% A la place de solve satisfy, on peut utiliser une stratégie telle que :
% solve :: int_search(x, first_fail, indomain_min, complete) satisfy;
```

Implantation MiniZinc ...suite

Une solution Minizinc avec optimisation du nombre de couleurs :

```
% Graphe G=(V, E)
% nombre de noeuds
int: n;

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

% ensemble des aretes (Deux noeuds par arête)
int: num_edges;
array[1..num_edges, 1..2] of V: E;

% Max couleurs=5 (On sait que 4 couleurs suffiront !)
int: nc = 5;

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

% u[c] = 1 veut dire : la couleur c est utilisée (affectée à au moins un noeud)
array[1..nc] of var 0..1: u;

% L'objectif : minimiser le nombre de couleurs
var int: obj = sum(c in 1..nc) (u[c]);

solve minimize obj;
```

Implantation MiniZinc ...suite

```
constraint
  % Contrôle des données : Pas de loop (circuit) dans le graphe
  forall(i in 1..num_edges) (
                E[i,1] != E[i,2]
  )
  ∧
  % Tout noeud a exactement une couleur
  forall(i in V) (sum(c in 1..nc) (x[i,c]) = 1)
  ∧
  % Les noeuds adjacents n'ont pas la même couleur
  forall(i in 1..num_edges, c in 1..nc) (
                x[E[i,1],c] + x[E[i,2],c] ≤ u[c]
  )
;

% Affichage des résultats
output [
  "obj: ", show(obj),"\n",
  "u: ", show(u), "\n",
] ++
[
  if j = 1 then "\n" ++ show(i) ++ ": " else " " endif ++
  show(x[i,j])
  | i in V, j in 2..nc
] ++ ["\n"];
```

Implantation MiniZinc ...suite

```
% Les données (le graphe des contraintes)
n = 13;
num_edges = 19;

% Le graphe des contraintes (couples de noeuds adjacents = incompatibles) :
E = array2d(1..num_edges, 1..2,
[1, 6,
1, 5,
1, 9,
1, 13,
2, 5,
2, 10,
2, 12,
3, 10,
6, 9,
6, 10,
6, 13,
6, 12,
5, 9,
5, 13,
9, 11,
8, 13,
8, 11,
8, 12,
10, 13, ]);
```

Implantation MiniZinc ...suite

Solution :

```
obj: 4          % 4 couleurs
    u: [1,1,1,1,0]          ← 4 couleurs utilisées
    1:  0 0 0 1 0           ←  4ème couleur affectée au noeud 1
    2:  0 0 0 1 0
    3:  1 0 0 0 0           ←  1ère couleur affectée au noeud 3
    4:  1 0 0 0 0
    5:  0 1 0 0 0
    6:  0 1 0 0 0
    7:  1 0 0 0 0
    8:  0 0 1 0 0
    9:  0 0 1 0 0
    10: 0 0 1 0 0
    11: 1 0 0 0 0
    12: 1 0 0 0 0
    13: 1 0 0 0 0
```

Remarques :

- De manière symétrique, on peut considérer le graphe des **compatibilités**. Dans ce cas, on constitue autant de paquets que de noeuds *compatibles* et on cherchera à affecter une même couleur à chaque paquet.
  - Par ex. le feu sera vert seulement pour ce groupe
  - Le feu passera au vert à tour de rôle pour chaque groupe.

Sudoku : modèle et code

- Plus le niveau de l'outil est élevé, plus le modèle et le code sont similaires :

```
include "globals.mzn";

array [1..9, 1..9] of var 1..9: sq;

predicate row_diff(int: r) = all_different (c in 1..9) (sq[r, c]);
predicate col_diff(int: c) = all_different (r in 1..9) (sq[r, c]);
predicate subgrid_diff(int: r, int: c) = all_different (i, j in 0..2) (sq[r + i, c + j]);

constraint forall (r in 1..9) (row_diff(r));
constraint forall (c in 1..9) (col_diff(c));
constraint forall (r, c in 1, 4, 7) (subgrid_diff(r, c));

sq = [| _ , _ , _ , _ , _ , _ , _ , _ , _ |
        _ , 6, 8, 4, _ , 1, _ , 7, _ |
        _ , _ , _ , _ , 8, 5, _ , 3, _ |
        _ , 2, 6, 8, _ , 9, _ , 4, _ |
        _ , _ ,7, _ , _ , _ , 9, _ , _ |
        _ , 5, _ , 1, _ , 6, 3, 2, _ |
        _ , 4, _ , 6, 1, _ , _ , _ , _ |
        _ , 3, _ , 2, _ , 7, 6, 9, _ |
        _ , _ , _ , _ , _ , _ , _ , _ , _ |];
solve satisfy;
output ["sq = ", show(sq)];
```

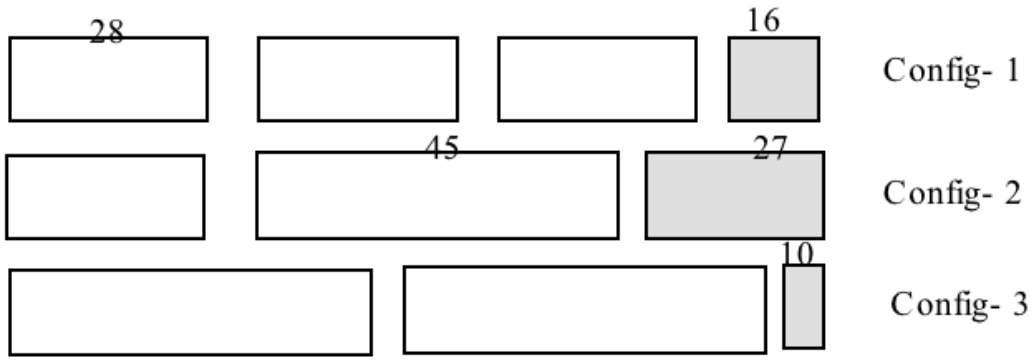
Sudoku : modèle et code ...suite

- **Solution** (la grille du départ est rappelée) :

```
          _, _, _, _, _, _, _, _, _
          _, 6, 8, 4, _, 1, _, 7, _
          _, _, _, _, 8, 5, _, 3, _
          _, 2, 6, 8, _, 9, _, 4, _
          _, _, 7, _, _, _, 9, _, _
          _, 5, _, 1, _, 6, 3, 2, _
          _, 4, _, 6, 1, _, _, _, _
          _, 3, _, 2, _, 7, 6, 9, _
          _, _, _, _, _, _, _, _, _

sq = [
          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
]
```

Problème de la découpe : CLP

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



- Une solution CLP

```
decoupe1(Chutes, Config_1, Config_2, Config_3) :-
    [Config_1, Config_2, Config_3] :: 0..36
    , Chutes = 16 * Config_1 + 27 * Config_2 + 10 * Config_3
    , 3* Config_1 + Config_2 = 36
    , Config_2 + 2* Config_3 = 24.
```

Problème de la découpe : CLP ...suite

Solution Minizinc avec la même modélisation :

- On pourrait prendre 36 barres et satisfaire directement la commande.
- Il faudra donc au maximum 36 de chaque configuration.

```
array[1..3] of var 0..36 : confs;
constraint 3*confs[1] + confs[2] >= 36;
constraint confs[2] + 2*confs[3] >= 24;
var int: chutes = 16*confs[1]+27*confs[2]+10*confs[3];

solve minimize chutes;

output["Conf1= ", show(confs[1]), " , Conf2= ", show(confs[2]), " ,
       Conf3= ", show(confs[3]), " , Chutes= ", show(chutes), "\n" ];
```

- Un test :
  Avec *Conf1= 12 , Conf2= 0 , Conf3= 12*
  ➙ Chutes= 312

• Il faut donc 12 des barres de Config-1, 12 de Config-3 et aucune de Config-2 et on aura 312 chutes (minimisés).

Exemple : Produits A-B

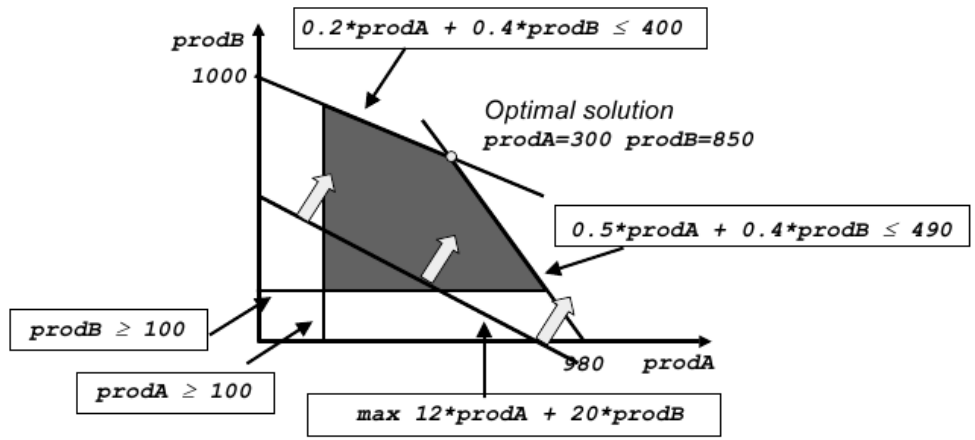
- 2 produits A et B à fabriquer sur deux machines M1 et M2.

  Les 2 produits ont besoin de ces 2 machines.
  La fabrication du produit A dure 12 minutes sur M1 et 30 sur M2;
  La fabrication du produit B dure 24 minutes sur M1 et 24 sur M2;
  Les ressources de M1 = 400 heures, 490 heures pour M2.
  Le profit d'un produit A est de 12 euros, 20 pour B.

- But : fabriquer au moins 100 unités de chaque produit en maximisant les bénéfices.
- **Modélisation** : versions OR (IP) et CP sensiblement identiques :

Zone hachurée = région *faisable*

```
max 12*prodA + 20*prodB  (Profit to be maximized)
s.t.
    0.2*prodA + 0.4*prodB ≤ 400  (Availability M1)
    0.5*prodA + 0.4*prodB ≤ 490  (Availability M2)
    prodA ≥ 100
    prodB ≥ 100                  (minimal quantity required)
    prodA, prodB integer
```



RO : un exemple BIP

Un problème de type entrepôt (Modélisation BIP) :

- Une entreprise construit une nouvelle usine à Lyon ou à Grenoble (ou les deux).
  - Un seul entrepôt mais là où on aura construit l'usine.
  - Les bénéfices et les couts sont donnés dans la table ci-dessous.
  - L'objectif est de maximiser les bénéfices.

Question oui/non	variablede 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.*

Modélisation :

- Variables de décision binaires : $\forall j = 1..4, x_j$ *binaire*
- Les contraintes de coût / ressource : $6x1 + 3x2 + 5x3 + 2x4 \leq 10$
- Un seul entrepôt : $x3 + x4 \leq 1$
- Entrepôt si Usine : $x3 \leq x1$, $x4 \leq x2$
- Objectif (bénéfices) : maximiser $9x1 + 5x2 + 6x3 + 4x4$

On évite les "=" sous Simplex!

$X_3 \implies X1$, V. la table de vérité !

RO : un exemple BIP ...suite

- On obtient le système (S)

$$
\begin{aligned}
maximize \quad & Z = 9x1 + 5x2 + 6x3 + 4x4 \\
s.t. \quad & 6x1 + 3x2 + 5x3 + 2x4 \leq 10 \\
& x3 + x4 \leq 1 \\
& -x1 + x3 \leq 0 \\
& -x2 + x4 \leq 0 \\
given \quad & x_j \ binaire \ \forall j = 1..4
\end{aligned}
$$

☞ Donner solution en exercice (v. cours 3).

N.B. : Cet exemple est traité dans l'addendum sur B&B.

Modélisation CP

Problème des paysans :

- 3 paysans : Modeste, Paul et Claude.
- 4 outils à louer (les 4 en même temps) :
  Taille-Haies (H), Motoculteur (M), Bêcheuse (B), Tracteur

- Contraintes d'ordre d'utilisation :
  Modeste (M) : H,T,B,M
  Paul (P) : T,M,H,B
  Claude (C) : M,T,B,H

Affecter ces outils aux paysans en respectant leurs contraintes et en minimisant le nombre de jours de location.

- On peut énumérer au moins 4 modélisations différentes de de l'exemple "Paysans".

☞ On sait qu'il faut au maximum 12 jours de location.

Modélisation CP ...suite

1. matrice **Paysan x outil**
   les cases contiennent un jour 1..12
   traduire les contraintes sur les lignes / colonnes
2. matrice **Outils x Jour**
   les cases contiennent un entir (nom ≡ numéro) de paysan
   exprimer les contraintes sur les lignes / colonnes
3. Formulation BIP :
   - matrice *bool* **Outil x jours** (1..12) de bool
     ➛ si 1 : tel outil est utilisé tel jour
   - matrice *bool* **paysans x jours**
     ➛ contient "1" si un paysan utilise un des outils tel jour
   - Il faut lier les 2 matrices.
4. un cube de variables *bool* : *cube C(o,p,j)* proche de la 3e méthode.
   C(o,p,j)=1 si .....
   Trouver les contraintes lignes / colonnes.
   ➛ Cette fois, pas besoin de lier les 2 matrice du modèle 3

☞ Codage de la 3e modélisation : ../..

Modélisation CP ...suite

```
/* N.B. : on peut utiliser "show" ou "fix" au besoin mais ce qui est affichée n'est pet être pas
   la solution finale (c'est une trace à un moment de la résolution).

Par contre, la fonction  trace(s,e) affiche le  string "s" avant d'évaluer l'expression "e"
  et renvoie sa valeur. Elle peut être utilisée n'importe où (n'importe quel contexte).
*/

include "globals.mzn";
include "increasing.mzn"; % Utilisé only pour le 1er paysans mais pour les 2 autres, je fais à la main

par int : max_jours = 12;
par int : nbPaysans = 3;
par int : nbOutils = 4;

array[1..nbPaysans, 1..max_jours] of var bool : Mat_paysans_jours;
array[1..nbOutils, 1..max_jours] of var bool : Mat_outils_jours;
array[1..nbPaysans, 1..nbOutils] of var 1..max_jours : Mat_paysans_outils;
var 1..max_jours : val_max_jours;  % Le max de val de jour à minimiser

/* Les outils H=taille haies, T=tronçonneuse,B=boucheuse, M=motoculteur
Les préférences
P1 : Modeste :< H , T , B , M >     (O1, O2, O3, O4)
P2 : Paul :< T , M , H , B >        (O2, O4, O1, O3)
P3 : Claude :< M , T , B , H >       (O4, O2, O3,O1)
*/

% trouve le jour où on a TRUE dans les 2 matrices.
function var int : jour_ou_on_a_true_dans_les_deux_matrices(int : p, int : o) =
  let {var int: x;
       constraint
          exists(j in 1..max_jours) (
            if (Mat_paysans_jours[p,j] /\ Mat_outils_jours[o,j]) then x=j else x=0 endif
          )
  } in  x ;
```

Modélisation CP ...suite

```
% contrainte sur la matrice Mat_paysans_jours
constraint
  forall(p in 1..nbPaysans)(
    sum([Mat_paysans_jours[p,j] | j in 1..max_jours])=nbOutils
  )
  /\
  forall(o in 1..nbOutils)(
    sum([Mat_outils_jours[o,j] | j in 1..max_jours])=nbPaysans
  )
  /\
  forall(j in 1..max_jours)(
    sum([Mat_paysans_jours[p,j] | p in 1..nbPaysans])= sum([Mat_outils_jours[o,j] | o in 1..nbOutils])
  )
  ;

/*
Rappel des préférences :
P1 : Modeste :< H , T , B , M >     (O1, O2, O3, O4)
P2 : Paul :< T , M , H , B >        (O2, O4, O1, O3)
P3 : Claude :< M , T , B , H >       (O4, O2, O3,O1)
*/

% Expression des contraintes à la main car les préférences sont très mélangées
constraint
  % Pour P1
  increasing([Mat_paysans_outils[1,o] | o in 1..nbOutils])

  % Pour P2
  /\ Mat_paysans_outils[2,2] < Mat_paysans_outils[2,nbOutils]
  /\ Mat_paysans_outils[2,nbOutils] < Mat_paysans_outils[2,1]
  /\ Mat_paysans_outils[2,1] < Mat_paysans_outils[2,3]

  % Pour P3
```

Modélisation CP ...suite

```
  /\ Mat_paysans_outils[nbPaysans,nbOutils] < Mat_paysans_outils[nbPaysans,2]
  /\ Mat_paysans_outils[nbPaysans,2] < Mat_paysans_outils[nbPaysans,3]
  /\ Mat_paysans_outils[nbPaysans,3] < Mat_paysans_outils[nbPaysans,1]
  ;

% Je relie la matrice  Mat_paysans_outils aux deux autres.
constraint
  forall(p in 1..nbPaysans) (
    forall(o in 1..nbOutils) (
       Mat_paysans_outils[p,o]=jour_ou_on_a_true_dans_les_deux_matrices(p,o)
    )
  );

% Contraintes sur la 3e matrice :
constraint
  forall(p in 1..nbPaysans) (
    alldifferent([Mat_paysans_outils[p,o] | o in 1..nbOutils])
  )
  /\
  forall(o in 1..nbOutils) (
    alldifferent([Mat_paysans_outils[p,o] | p in 1..nbPaysans])
  )
  ;

% Optimisation}
constraint
  % on peut mettre directement max sur la matrice: max(Mat_paysans_outils)
  val_max_jours=max([Mat_paysans_outils[p,o] | p in 1..nbPaysans, o in 1..nbOutils])
  ;

solve minimize val_max_jours ;

  output["\tJ" ++ show(i) | i in 1..max_jours ]
    ++ ["\n"++ "P1\t"]
```

Modélisation CP ...suite

```
    ++ [show(Mat_paysans_jours[p,j]) ++ "\t" ++
        if j=max_jours then "\nP"++show(p+1)++"\t" else "" endif
        | p in 1..nbPaysans, j in 1..max_jours]
    ++ ["\n----------------------------------\n"]
    ++ ["\tO" ++ show(i) | i in 1..nbOutils ]
    ++ ["\n"++ "P1\t"]
    ++ [show(Mat_paysans_outils[p,o]) ++ "\t" ++ if o=nbOutils then "\nP"++show(p+1)++"\t" else "" endif| p in
    ;

/* TRACE
OK pour solveur CPX (begonia 42.1, CPX est pris dans minizinc 1.6, tester sous 42.2)
Le solveur par défaut donne une sol puis cherche.

Running paysans3.mzn
          J1      J2      J3      J4      J5      J6      J7      J8      J9      J10     J11     J12
P1      false   true    true    true    true    false   false   false   false   false   false   false
P2      true    false   true    true    true    false   false   false   false   false   false   false
P3      true    true    true    false   true    false   false   false   false   false   false   false
----------------------------------
          J1      J2      J3      J4      J5      J6      J7      J8      J9      J10     J11     J12
O1      false   true    false   true    true    false   false   false   false   false   false   false
O2      true    true    true    false   false   false   false   false   false   false   false   false
O3      false   false   true    true    true    false   false   false   false   false   false   false
O4      true    false   true    false   true    false   false   false   false   false   false   false
----------------------------------
          O1      O2      O3      O4
P1      2         3       4       5
P2      4         1       5       3
P3      5         2       3       1
*/
```

Modélisation CP ...suite

Solution optimale :

Paysans / Outil	O1	O2	O3	O4
P1	2	3	4	5
P2	4	1	5	2
P3	5	2	3	1

Unification CP / CLP : exemple

Exemple n-queen sera traduit en *MiniZinc*

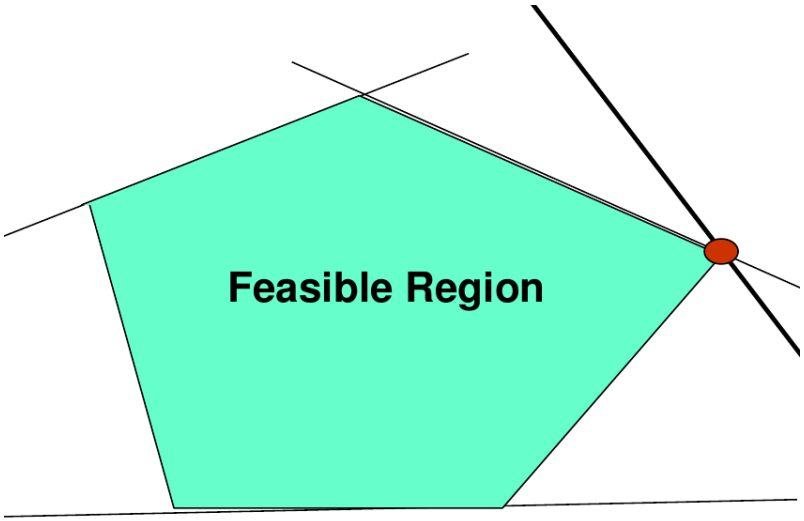
```
array[0..9] of var 0..9: queens;
int: n;

n = 10;

queens(N) :- queens(N, 0).

queens(N, I) :-
  I < N,
  queens(N, I, I + 1),
  queens(N, I + 1).
queens(N, N).

queens(N, I, J) :-
  J < N,
  queens[I] != queens[J],
  queens[I] + I != queens[J] + J,
  queens[I] - I != queens[J] - J,
  queens(N, I, J + 1).
queens(N, _, N).

include "prolog.plz"; %<<<---

labeling_list([], _, _).
labeling_list([H | T], Min, Max) :-
    between(Min, H, Max),
    labeling_list(T, Min, Max).

:- queens(n), labeling_list(queens, 0, n - 1).

output [show(queens)];
```

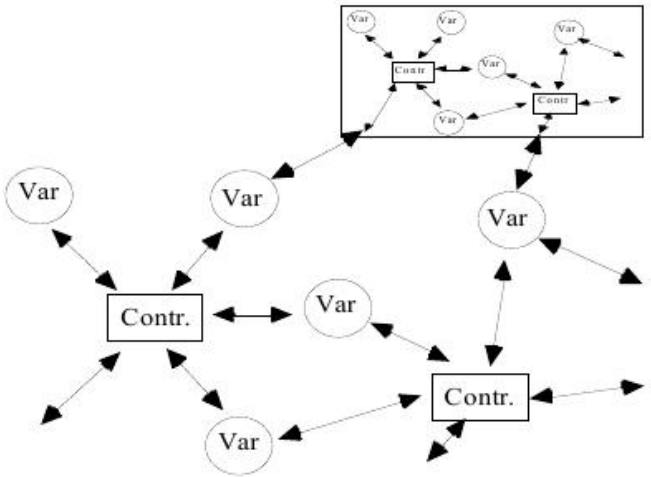
Quelques outils

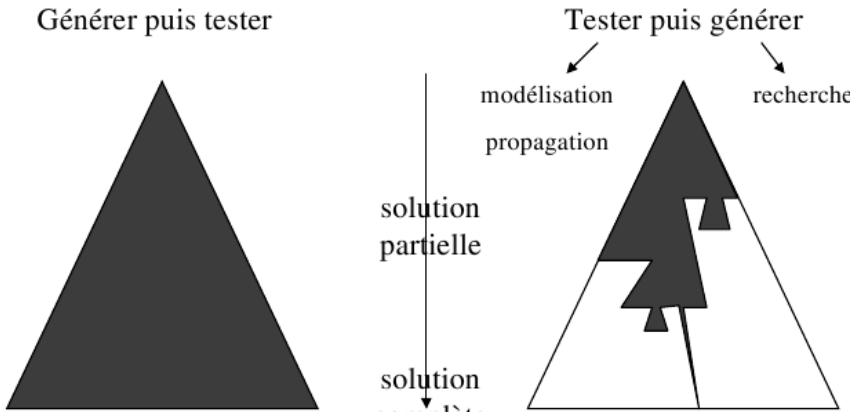
Outils LP/CP/CLP:

- LpSolve
- Glpk
- Plugins pour Excel
- Autres environnements de Programmation Mathématique / Linéaire (et non linéaires)
- MiniZinc
- GAMS (très intéressant, LP, NLP, etc.)
- CPLEX / AMPL
- OPL / KNITRO (non linéaire !)
- Eclipse / BProlog / Picat / GProlog (Solveurs de contraintes + BT)
- PIII, PIV (Solveurs de contraintes + BT)

Comparaison des espaces de recherche







- En haut à gauche : espace de solutions
- En haut à droite :
  graphe de contraintes (CSP)
- En bas à gauche :
  optimisation CSP (CLP)

☞ Frontières LP/MP/CSP/CLP flues !

Aperçu du CP (domaine fini)

- Modélisation :
  - Variables de domaine fini (e.g. X :: 1..1000)
    Mais de type quelconque (l'important est la finitude / énumération)
    → Les réels sous forme d'intervalles.
  - Contrainte (Arithmétiques / Symboliques) sur des variables
  - Contrainte Globales (Arithmétiques / Symboliques)

- Résolution :
  - Algos de propagation exprimés dans les contraintes mêmes (NC, AC, PC)
    → e.g. : $NC : X < 2, X > 2$, $AC : X > Y$ et $PC : X > Y, Y > Z$
  - Principalement NC et AC (PC coûte cher)
    ➙ Propagation plus complexes pour les variables globales
  - Stratégies de recherche (parcours graphes Descendant, Ascendant, ...)
    ➙ B & B
    ➙ Look Ahead, Forward Check (+ retours arrières pour CLP),...

Aperçu du CP (domaine fini) ...suite

CSP/CP : Techniques de Consistance (Noeud, Arc et Chemin)

- Il est possible de construire un solveur (complet) avec ces consistances + BT.
  ➙ Soit $X = \{X_1, X_2, X_3\}$, $D = \{D_1, D_2, D_3\}, D_i = \{1, 2, 3\}$,
  $Z = \{X_1 > X_2 > X_3\}$

- **Node Consistance :**
  - Aucune simplification, les variables gardent leur domaine.
  - Les contraintes des domaines sont vérifiées.
  - Exemple : X *in* $1..3, X > 3$ n'est pas valide.

- **Arc Consistance :**
  - P. ex. si $X_1 > X_2 \quad \rightarrow \quad D$ devient : $D_1 = \{2, 3\}, D_2 = \{1, 2\}$.
  - Puis indépendamment pour X2 et X3.

- **Path Consistance** (coûteux) :
  ➙ On peut simplifier et D devient : $D_1 = 3, D_2 = 2, D_3 = 1$

- Exemples et détails plus loin.

Aperçu du CP (domaine fini) ...suite

- Modélisation CP :
  - La modélisation se fait en termes du triplet V, D, C
  - La fonction objective est une contrainte globale.

- Variables : entités du problème,
- Domaines : valeurs possibles,
- Contraintes : relations entre variables,
- Fonction objective = critère d'optimisation

- Dans l'exemple de coloration :

Variables X_1, ... X_n (les noeuds)

Domaines Couleurs $\in$ {rouge, bleu, vert ... }

Contraintes $X_i \neq X_j \quad \forall i \neq j$ pour certains i et j (noeuds voisins)

Objectif minimiser nombre couleurs

Les Contraintes dans CP

- Affectation vs. Équation,
  ➙ Test vs Contrainte.

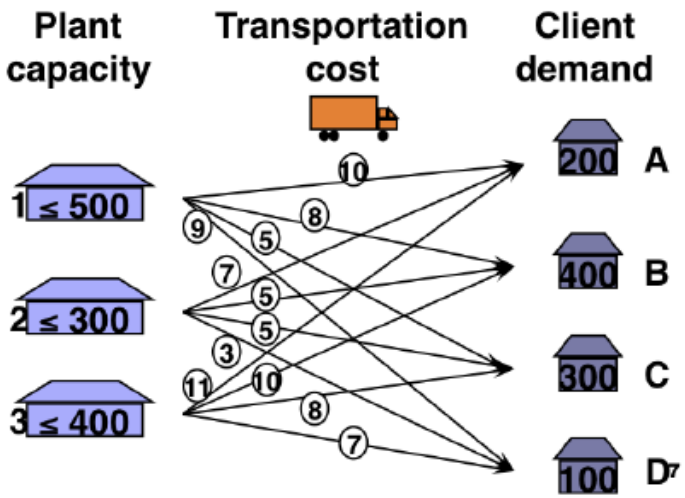
- **Contraintes Arith** : $=, \neq, >, <, \geq, \leq$
  ➙ Avec AC, PC

- **Contraintes Symboliques** / globales :
  ➙ Avec des algorithmes de propagation plus puissants (que AC)
  - **alldifferent**($[V_1, V_2, ..., V_n]$)
  - **element**(N, $[V_1, V_2, ..., V_n]$, Valeur)
  - **cumulative** ($[S_1, S_2, ..., S_n]$,$[D_1, D_2, ..., D_n]$, $[R_1, R_2, ..., R_n]$,Lim_resses)
    ➙ S_i : start time, D_i : Duration, R_i : ressource,
  - **disjonctive** : $C_1 \backslash/ C_2 \backslash/ \ldots \backslash/ C_n$
  - ...

Exemple Entrepôts

- En CP, on peut exprimer les contraintes usuelles de LP (cas des réels).
- Un exemple : entrepôts et clients avec minimisation de "Cout".

```
% A[1] : ce que A reçoit de l'entrepôt 1.
% B[3] : ce que B reçoit de l'entrepôt 3.
...
array[1..3] of var float : A;
array[1..3] of var float : B;
array[1..3] of var float : C;
array[1..3] of var float : D;
var float : Cout;
constraint
      forall([A[i]>= 0.0 | i in 1..3])
   /\ forall([B[i]>= 0.0 | i in 1..3])
   /\ forall([C[i]>= 0.0 | i in 1..3])
   /\ forall([D[i]>= 0.0 | i in 1..3])
   /\ A[1] + A[2] + A[3] = 200.0                    % demandes
   /\ B[1] + B[2] + B[3] = 400.0
   /\ C[1] + C[2] + C[3] = 300.0
   /\ D[1] + D[2] + D[3] = 100.0
   /\ A[1] + B[1] + C[1] + D[1] <= 500.0            % capacités
   /\ A[2] + B[2] + C[2] + D[2] <= 300.0
   /\ A[3] + B[3] + C[3] + D[3] <= 400.0
   /\ Cout = 10.0*A[1] + 7.0*A[2] + 11.0*A[3] +
             8.0*B[1] + 5.0*B[2] + 10.0*B[3] +
             5.0*C[1] + 5.0*C[2] + 8.0*C[3] +
             9.0*D[1] + 3.0*D[2] + 7.0*D[3] ;
```



Résultats (solveur mip de Minizinc) :

Cout = 6600.0
A= [100.0, 0.0, 100.0]
B= [100.0, 300.0, 0.0]
C= [300.0, 0.0, 0.0]
D= [0.0, 0.0, 100.0]

```
solve minimize(Cout);
output["Cout= ", show(Cout), show(A), , show(B), show(C), show(D)];
```

Exemple Pierres : Stones

- Une pierre de 40 kg se brise en 4 morceaux.
- Avec ces morceaux, on peut peser tous les objets de 1 a 40 kg sur une balance à 2 plateaux.
- Quel est le poids de chacun des 4 morceaux ?

Une Modélisation :

- 4 variables (entiers) pour les 4 morceaux (soit L=$[P_1, P_2, P_3, P_4]$)
  - ➙ $P_i \in 1..40, \quad \sum P_i = 40.$
- Les 4 poids devraient être différents (plus pratique !) : $P_i \neq P_j, i \neq j$.
- Tous les poids 1..40 sont mesurables par les 4 morceaux.

- On regroupe l'ensemble des contraintes :

$L = P1, P2, P3, P4,$
$L :: 1..40,$ % notation générique du domaine
$P_1 + P_2 + P_3 + P_4 = 40,$
$P1 \leq P2, P2 \leq P3, P3 \leq P4,$ % pour éviter la symétrie
$tous\_les\_poids\_mesurables(L, 1..40)$

Exemple Pierres : Stones ...suite

tous_les_poids_mesurables$(L, 1..N)$:

Pour $X = 1..40$, vérifier que X est mesurable par $L = P1, P2, P3, P4$.

- Pour chaque mesure, on envisage les deux plateaux de la balance.

  ➙ On associe une variable B_i à P_i avec $B_i \in \{-1, 0, +1\}$:

  - $B_i = -1$: P_i sur le plateau de gauche
  - $B_i = +1$: P_i sur le plateau de droite
  - $B_i = 0$: P_i ne participe pas pour mesurer le poids X

- $\forall X \in 1..40$, *peser* X *avec* $P1, P2, P3, P4$

$$X = \sum_{i=1}^{4} B_i P_i$$

$$B_i :: -1..1.$$

- Solution : L = [1, 3, 9, 27].
- Optionnel : une matrice[1..40][1..4] de -1..1 conserve l'agencement des pierres.

Exemple Pierres : Stones ...suite

- Une solution Minizinc

```
include "increasing.mzn";   % pour casser la symétrie

par int : masse = 40;       % masse est un "par"—amètre (vs. var)
par int : nombre_pierres = 4;

set of int : a_peser = 1..masse;

array[1..nombre_pierres] of var 1..masse : pierres;
array[a_peser,1..nombre_pierres] of var -1..1 : combinaisons;

constraint
    % On brise la symétrie du problème
    increasing(pierres)
  /\
    % La somme des masses des pierres est egale a la masse totale recherchee
    sum(p in 1..nombre_pierres)(pierres[p]) = masse
  /\
    % Chacune des masses à peser est obtenue par addition
    %    ou soustraction des masses de pierres
    forall(m in a_peser)(
        sum(p in 1..nombre_pierres)(pierres[p]*combinaisons[m,p]) == m
    )
  ;

solve satisfy;
output[show(pierres),"\n",show(combinaisons)];
```

Exemple Pierres : Stones ...suite

- Test :

```
% minizinc stone.mzn
%   [1, 3, 9, 27]
%   [1, 0, 0, 0,
    -1, 1, 0, 0,     % pour peser 2kg : la pierre de 3kg d'un côté, de 1kg de l'autre
     0, 1, 0, 0,
     1, 1, 0, 0,
    -1,-1, 1, 0,     % pour peser 5kg : 9kg d'un côté, 1kg et 3kg de l'autre
     0,-1, 1, 0,
     1,-1, 1, 0,
    -1, 0, 1, 0,
     0, 0, 1, 0,
     1, 0, 1, 0,     % pour peser 10kg : 9kg et 1kg d'un même côté

    -1, 1, 1, 0,      0, 1, 1, 0,      1, 1, 1, 0,      -1,-1, -1, 1,
     0,-1,-1, 1,      1,-1,-1, 1,     -1, 0,-1, 1,       0, 0, -1, 1,
     1, 0,-1, 1,     -1, 1,-1, 1,      0, 1,-1, 1,       1, 1, -1, 1,
    -1,-1, 0, 1,      0,-1, 0, 1,      1,-1, 0, 1,      -1, 0,  0, 1,
     0, 0, 0, 1,      1, 0, 0, 1,     -1, 1, 0, 1,       0, 1,  0, 1,
     1, 1, 0, 1,     -1,-1, 1, 1,      0,-1, 1, 1,       1, -1, 1, 1,
    -1, 0, 1, 1,      0, 0, 1, 1,      1, 0, 1, 1,      -1,  1, 1, 1,
     0, 1, 1, 1,
     1, 1, 1, 1     % 40 Kilos : toutes les pièrres utilisées
    ]
```

➙ La matrice combinaisons contient 40 lignes et 4 colonnes (dans -1..1).

- Voir plus loin pour la modélisation CLP.

Résolution CP

La résolution d'un problème CSP/CP s'appuie sur :
- La consistance
- La propagation de contraintes
- Une stratégie de recherche (énumération en CLP)
- Back-Track (en CLP)

- Consistance & existence de solution
  - NC/ AC/PC
  - L'ensemble des contraintes (*store*) est-il consistant ?
  - Est-ce qu'une solution existe ?

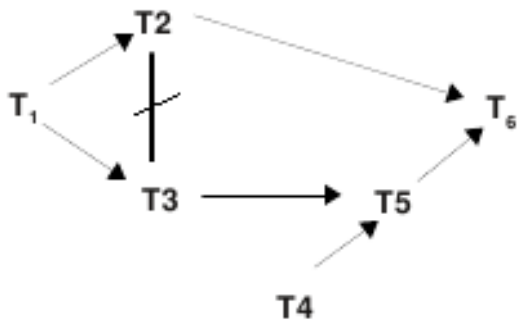
- Propagation : mécanismes d'inférence
  - Suppression de valeurs dans les domaines
  - Inférence de nouvelles Contraintes

- Recherche : stratégie de branchement (suivi de propagation)
  - Choix de variable,
  - Choix de valeur, ...(énumération)
  - Autres (selon le problème)

Résolution CP ...suite

Exemple : *PERT* (avec propagation de contraintes)

- On veut les dates de début des tâches.
- La propagation des contraintes :
  Si on propage les contraintes (directement puis par transitivité = Arc/Path-consistancy), les domaines des variables se modifient.
- Propagations des contraintes de l'exemple :
  Domaines : $T1, T2, T3, T4, T5, T6 \in 1..5$
  Contraintes : $T1 < T2, T1 < T3, T2 \neq T3,$
  $T2 < T6, T3 < T5, T4 < T5, T5 < T6.$



$T_1 < T_2$
$T_1 < T_3$
$T_2 \not< T_3$
$T_2 < T_6$
$T_3 < T_5$
$T_4 < T_5$
$T_5 < T_6$

- Les propagations donnent : $T1 :: 1..2,\ T2 :: 2..4,\ T3 :: 2..3,\ T4 :: 1..3,$
  $T5 :: 3..4,\ T6 :: 4..5$
- Si on pose $T2 = 2$: déclenche d'autres propagations
  ➟ $T1 = 1, T2 = 2, T3 = 3, T4 :: 1..3, T5 = 4, T6 = 5$

CP : Aperçu d'optimisation

- Dans certains problèmes, on n'est pas intéressé par connaître une solution faisable mais plutôt par une solution optimale selon certains critères
  - Ex. : dans coloration, on veut savoir le nombre min de couleurs sans connaître les couleurs affectées aux noeuds.

- Énumération (CLP) : pas très efficace (en tout cas, si c'est fait trop tôt)
  - Recherche d'une solution faisable
  - Choix de la "meilleure" valeur
  - Options d'énumération : les plus contraintes, domaine le plus petit, etc....

- Les outils CP englobent en général une forme simplifiée de B&B
  - A chaque solution trouvée avec un cout C^*, contraindre l'arbre de recherche à ne développer que les solutions où le cout risque d'être "meilleur" que C^*.
  - Et si un meilleur C est trouvé alors remplacer C^* par ce cout.
  - B & B : trouver une solution avec $cout < cout*$

- La traduction de certains schémas en leur équivalent relève de l'optimisation.

Programmation Mathématique

• La Programmation avec Contraintes libère de certaines considérations opératoires pour nous concentrer sur le modèle.

• En Programmation Mathématique, on traduit les expressions de plus haut niveau (P. ex. , les contraintes globales) en un niveau plus simple en utilisant des schémas de traduction (utilisés également pour optimiser un code CP).

• **Quelques exemples** (voir plus loin + BE pour la méthode) :

- Si on veut : $f(x) \leq O \ \vee \ g(x) \leq 0$
  $f(x) \leq M.y, \ g(x) \leq M.(1-y)$ — $y : bool, M : un\ grand\ entier$
- Si on veut : $f(x) > 0 \implies g(x) \geq 0$ — on sait que $A \implies B \equiv \neg A \vee B$
  $-g(x) \leq M.y, f(x) \leq M.(1-y)$ — $y = 0$ si $A = vrai$
- Si on veut $A \neq B$
  $A \neq B \equiv (A > B) \vee (B > A)$ — On sait faire pour la disjonction.

  Soit $A, B \in L..U$ et $Z \in \{0,1\}$:
  Alors : $Z * (U + 1 - B) + B > A > (L-1) + Z * (B - (L-1))$
  ➛ Si $Z = 0$, $on\ a\ B > A > L-1$; $Si\ Z = 1$, $on\ a\ U + 1 > A > B$

Programmation Mathématique ...suite

- Si on veut : $x, y \in 1..n, x \neq y$

  var 1..n : x; *var 1..n : y;*
  array[1..n] of var 0..1 : tx; *% La valeur prise par x*
  *constraint sum(i in 1..n)(i*tx[i]) = x*
  $\wedge$ *sum(i in 1..n) (tx[i]) = 1*
  % ... idem pour y.
  $\wedge$ *forall(i in 1..n)(tx[i] + ty[i] <= 1);* *% Pour modéliser l'inégalité*

- Si on veut : $element(i, [a_1, a_2, ..., a_n], x)$ qui veut dire : $\exists i, x = a_i$

  OU $(i = j) \implies (x = a_j)$ (notons $a_j = a[j]$)

  array[1..n] of var 0..1 : ti;
  constraint x = sum(j in 1..n)(ti[j] aj);*

  Exemple : une contrainte comme $t_j \leq 5$ peut être implantée par :

  $$Z \leq 5 \wedge element(j, (t_1, \cdots, t_n), Z)$$

Programmation Mathématique ...suite

- Si on veut : *array[1..n] of var 1..m : tab;*
  constraint alldifferent(tab); % équivalent à *all_different*
  ➛ On sait faire...

- Soit la contrainte $abs(f(x)) \geq m$ similaire à $abs(x) \geq y \equiv (x \geq y \vee -x \geq y)$
  ◦ 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 $-m >= f(x) >= L$ (II)
  Et donc : $z * (U + m) - m \geq f(x) \geq L + z * (m - L)$ (III)

- ...

• Les variables 0..1 permettent des modélisations intéressantes.

☞ Les outils CP / CLP (p. ex. Minizinc) disposent des contraintes de haut niveau mais on peut avoir parfois recours à ces techniques d'optimisation.

Rappel des Techniques LP

Vocable *Programmation Mathématique* pour désigner les modèles de résolution des problèmes d'optimisation.

- Programmation linéaire : **LP**
- Programmation entière (et bool) : **IP, BIP**
- Mixte Programming : **MP, MIP**
- Non linéaire : **NLP**

- Résolution par l'algorithme **Simplex** (de base)
- **Les techniques**
  - Branch and Bound (B & B)
  - Plans de coupe
  - Génération de colonnes
  - ...

Programmation Entière (IP)

- Méthode standard venant du domaine d'optimisation combinatoire.
- Forme générale d'un système IP (PLNE) :

$$
\begin{aligned}
min \quad & Z = \sum_{j=1}^{n} C_j X_j \\
s.t. \quad & \sum_{j=1}^{n} a_{ij} X_j = b_i, \qquad i = 1..m \\
& x_j \geq 0, \qquad j = 1..n \\
& x_j \ \textit{entier} \rightarrow \text{peut rendre le problème NP-complet}
\end{aligned}
$$

- Un certain style (norme) à respecter :
  - *min* pour l'objectif et $\geq$ entre variables / constantes
  - *max* pour l'objectif et $\leq$ entre variables / constantes

N.B. : On exprime la maximisation par la négation de la minimisation

Programmation Entière (IP) ...suite

Relaxation linéaire (voir exemples plus loin) :

- On écarte la contrainte d'intégralité (*integer*)
  ➙ Le système **IP-relaxé** devient

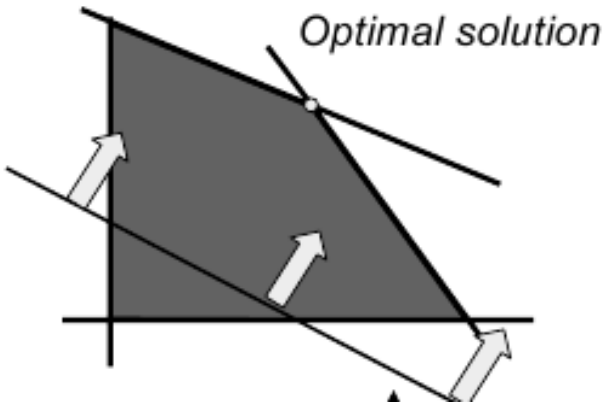
$$
\begin{aligned}
& min && Z = \sum_{j=1}^{n} C_j X_j \\
& s.t. && \sum_{j=1}^{n} a_{ij} X_j = b_i, \qquad i = 1..m \\
& && x_j \geq 0, \qquad j = 1..n
\end{aligned}
$$

enlever x_j entier ➙ peut rendre le problème NP-complet

- La complexité de *IP-relaxé* est réduite au niveau polynomial.
- La méthode la plus utilisée est **Simplex**, même avec une complexité exponentielle (cas pire) du système à résoudre.

Solveur LP : Simplex

$$
\begin{aligned}
min \quad & Z = \sum_{j=1}^{n} C_j X_j \\
s.t. \quad & \sum_{j=1}^{n} a_{ij} X_j = b_i, \qquad i = 1..m \\
& x_j \geq 0, \qquad j = 1..n
\end{aligned}
$$



La ligne ⇑ ⇑ qui glisse vers la solution optimale représente la fonction objective.

- L'ensemble des contraintes définit un *polytope*
- La solution optimale est sur l'un des sommets

La méthode Simplex débute par un sommet et va vers le sommet adjacent avec une meilleure valeur pour la fonction objective.

Primal et Dual

• La solution à *IP-relaxé* est un ensemble de valeurs pour les variables respectant les contraintes linéaires tout en optimisant la fonction objective.
  ➙ Mais cette solution viole la contrainte d'intégralité initiale.

• Pour faciliter la solution du *Primal*, on peut avoir recours à son *Dual*.

• Chaque système IP initial (Primal) a un système Dual qui lui est associé;
  ➙ Les variables du Dual correspondent aux contraintes du Primal et les *contraintes du Dual* correspondent aux *variables du Primal* (IP).

Intérêt du dual :
• Le Dual et le Primal ont les fonctions objectives opposées mais ont le même extrémum.

• Le choix entre Primal / Dual :
  ◦ On avance la résolution du Primal et du Dual en parallèle **ou**
  ◦ On choisit celui qui est plus simple à résoudre
  → En particulier si seul l'extremum importe.

Primal et Dual ...suite

- **Signification du Dual :**

  Puisque les contraintes dans Primal vont correspondre aux variables dans Dual et les variables du Primal aux contraintes dans Dual :

  - On peut calculer l'effet de la variation d'une valeur :

  ➙ C-à-d. dans $Ax = b$ du Primal, on peut connaitre le cout d'une variation de la valeur de b.

  Par exemple : dans $Ax = b$, si b correspond à une quantité de stock (pour un marchand avec ses coûts de fabrication et ses profits à la vente), on peut savoir quel serait l'effet de $b - 1$ sur ses coûts/profits.

- Si la fonction objective du Primal est $Min\ Z$ (resp. $Max\ Z$), la solution du Dual permet de connaître la borne inf (resp. sup) de Z.

  - Pour l'objectif $min\ Z = ...$ dans le Primal, le Dual dira $Z \geq \alpha$
    → la valeur minimale de Z ne sera pas $< \alpha$.
  - De même pour le cas $max\ Z$ ....

Primal et Dual ...suite

Rappel de l'intérêt du Dual :

- le Dual et le Primal ont le même extrémum.
- Si seul l'extrémum nous intéresse, on utilise le plus simple des deux.

Un exemple pratique d'utilisation :

domaine de produits fabriqués à partir de matières premières.

- Quel serait le prix de vente de ces matières premières si on veut obtenir les mêmes profits sans fabriquer les produits (sans subir le cout de la fabrication / vente).
- Celui qui fabrique veut maximiser ses profits et celui qui achète les matières premières veut minimiser le cout de son achat.

Formulation Primal-Dual

Correspondance Primal-Dual pour un système général $Ax = b$

- Primal

$$\begin{aligned} min \quad & cx \\ s.t. \quad & Ax \geq b, \quad x \geq 0 \end{aligned}$$

- Dual

$$\begin{aligned} max \quad & by \\ s.t. \quad & Ay \leq c, \quad y \geq 0 \end{aligned}$$

- N.B. : mise sous la forme standard (normale) :
  ➙ On peut transformer $Max\ cx$ en $Min\ -cx$
  Et de ré-écrire une contrainte $Ax \geq b$ en $-Ax < -b$

- Min avec $\geq$ ou Max avec $\leq$ sont les formes préférées (dites *standard*).

Formulation Primal-Dual ...suite

Une formulation plus précise :

Pour un Primal $\mathfrak{P}$:

$$\begin{aligned} min \quad & Z = cx \\ s.t. \quad & Ax \geq b, \\ & x \geq 0 \end{aligned}$$

Le Dual $\mathfrak{D}$ sera :

$$\begin{aligned} max \quad & v \\ s.t. \quad & (Ax \geq b) \vdash^{Q} (cx \geq v) \\ & v \in \mathbb{R}, Q \in \mathfrak{D} \end{aligned}$$

➙ $(cx \geq v)$: le Primal minimisera $Z = cx$ mais on veut connaître v telle que $Z = cx \geq v$ (Z minimale mais pas plus petite que v)

☞ Pour la fonction objective du Primal $Min\ Z = cx$, la solution du Dual permet de connaître v la borne inf de Z.

→ Pour $min\ Z = cx$ dans Primal, Dual dira $Z \geq v \rightarrow cx$ ne sera pas $< v$.

➥ Donc, Dual nous informe sur l'extrémum de Z : v.

➙ Q : les variables de *preuve* dans Dual à valeur dans une combinaison des contraintes d'inégalité non négatives présentes dans $Ax \geq b$ $\cdots \rightsquigarrow$

Primal-Dual : Détails d'un exemple

Soit le problème :

$$\begin{aligned} Min \quad & Z = 2x + y \\ s.t. \quad & x + y \geq 1 \\ & x - y \geq -2 \\ & -x \geq -2 \end{aligned}$$

- On veut une estimation rapide du min de Z sans devoir résoudre le système.

- L'idée est de manipuler les équations de sorte qu'une borne inférieure de Z puisse être trouvée.
  - Ceci est fait en additionnant différents multiples des contraintes jusqu'à retrouver (la partie droite de) la fonction objective.
  - En plaçant Z entre sa borne du Primal et celle du Dual, on pourra alors déduire que la partie droite de la fonction objective est une borne inf de Z.

Primal-Dual : Détails d'un exemple ...suite

• Par exemple, si on additionne (3 × la 1ère contrainte) plus (2 × la seconde) plus (3 × la troisième), on obtient :

$$2x + y \geq -7$$

➙ Ceci nous dit que notre $Z \geq -7$:

➙ Z ne sera pas plus petite que -7.

$$\begin{array}{ll} Min & Z = 2x + y \\ s.t. & x + y \geq 1 \\ & x - y \geq -2 \\ & -x \geq -2 \end{array}$$

• Le but de ces manipulations (judicieusement choisies) est de retrouver la fonction objective $(2x + y)$.

➙ On Utilise les coefficients positifs pour ne pas modifier le sens des inégalités.

☞ Au lieu d'utiliser $(3, 2, 3)$, on peut utiliser $(2, 1, 1)$ et obtenir $2x + y \geq -2$.

➙ $Z \geq -2$: cette binf est meilleure que -7

☞ Idem avec $(a1, a2, a3) = (3/2, 1/2, 0)$: on obtient $Z = 2x + y \geq -1/2$

➙ On n'aura pas une meilleure estimation que $-1/2$.

☞ **Il faut donc une méthode.**

Formulation Dual

Rappels : concrètement

• Pour un programme linéaire général :

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

Le dual sera :

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

• **Symétriquement :**

➙ Si dans Primal, on a : $Max\{C^T x | Ax \leq b, x \geq 0\}$
Dans Dual, on aura : $Min\{b^T \alpha | A^T \alpha \geq c, \alpha \geq 0\}$

☞ α (ou Q ci-dessus) représente ici le vecteur des variables de la forme Duale.

Formulation Dual ...suite

Exemple : le système Primal vu ci-dessus :

$$\begin{array}{ll} Min & Z = 2x + y \\ s.t. & x + y \geq 1 \\ & x - y \geq -2 \\ & -x \geq -2 \end{array}$$

Notre Primal par la notation matricielle :

$$A \begin{bmatrix} x \\ y \end{bmatrix} \geq \begin{bmatrix} 1 \\ -2 \\ -2 \end{bmatrix}$$

Et, on a du Primal :

$$A = \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ -1 & 0 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 1 & -1 \\ 1 & -1 & 0 \end{bmatrix}$$

Et

$$c^T = \begin{bmatrix} 2 & 1 \end{bmatrix}$$

$$b^T = \begin{bmatrix} 1 & -2 & -2 \end{bmatrix}$$

Rappel : pour un Primal

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

On obtient le Dual

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

Le système Dual de notre exemple sera :

$$A^T \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \leq \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Et donc on aura le Dual :

$$\begin{array}{ll} Max & Z' = 1a_1 - 2a_2 - 2a_3 \\ s.t. & a_1 + a_2 - a_3 \leq 2 \\ & a_1 - a_2 \leq 1 \\ & a_1, a_2, a_3 \geq 0 \end{array}$$

- La résolution du Dual donne $Z \geq -0.5$ pour $a_1 = 3/2,\ a_2 = 1/2, a_3 = 0$

→ C'est le trio $(3/2, 1/2, 0)$ trouvé ci-dessus.

Un autre exemple

Soit LP Primal :

$$\begin{aligned} Max \quad & Z = 2x1 + 3x2 - x3 \\ s.t. \quad & x1 + x2 \leq 3 \\ & -x1 + 2x3 \geq -2 \\ & -2x1 + x2 - x3 = 0 \\ & x_i \geq 0, x_i \in \mathbb{R} \end{aligned}$$

- La mise sous forme standard donne (Primal) :

  ➙ Noter la 3e contrainte (= réécrit en $\leq$)

$$\begin{aligned} Max \quad & Z = 2x1 + 3x2 - x3 \\ s.t. \quad & x1 + x2 \leq 3 \\ & x1 - 2x3 \leq 2 \\ & -2x1 + x2 - x3 \leq 0 \\ & x_i \geq 0, x_i \in \mathbb{R} \end{aligned}$$

Un autre exemple ...suite

Rappel : le Primal :

$$\begin{aligned} Max \quad & Z = 2x1 + 3x2 - x3 \\ s.t. \quad & x1 + x2 \leq 3 \\ & x1 - 2x3 \leq 2 \\ & -2x1 + x2 - x3 \leq 0 \\ & x_i \geq 0, x_i \in \mathbb{R} \end{aligned}$$

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -2 \\ -2 & 1 & -1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & -2 \\ 1 & 0 & 1 \\ 0 & -2 & -1 \end{bmatrix}$$

$$b^T = \begin{bmatrix} 3 & 2 & 0 \end{bmatrix}$$

$$c^T = \begin{bmatrix} 2 & 3 & -1 \end{bmatrix}$$

Un autre exemple ...suite

- Si LP Primal est de la forme $Max\{c^T x | Ax \leq b, x \geq 0\}$
  ➞ La forme Duale sera: $Min\{b^T y | A^T y \geq c,\ y_i \geq 0\}$

☞ *Max* et *Min* symétriques.

- Pour notre LP (standardisé et toute inéquation sous forme $\leq$ avec *Max*) :

$$\begin{aligned} Max \quad & Z = 2x1 + 3x2 - x3 \\ s.t. \quad & x1 + x2 \leq 3 \\ & x1 - 2x3 \leq 2 \\ & -2x1 + x2 - x3 \leq 0 \\ & x_i \geq 0, x_i \in \mathbb{R} \end{aligned}$$

- Le système Dual (standardisé sous forme $\geq$ avec *Min*) :

$$\begin{aligned} Min \quad & Z' = 3y_1 - 2y_2 \\ s.t. \quad & y_1 - y_2 - 2y_3 \geq 2 \\ & y_1 + y_3 \geq 3 \\ & 2y_2 - y_3 \geq -1 \\ & y_i \geq 0, y_i \in \mathbb{R} \end{aligned}$$

Un autre exemple ...suite

- Le code en Minizinc :

```
% Exemple Primal Dual

array[1..3] of var float : x;
array[1..3] of var float : y;

constraint
      forall (i in 1..3) (x[i] >= 0.0)
  /\  forall (j in 1..3) (y[j] >= 0.0)
  ;

constraint
             x[1] + x[2]                 <= 3.0
  /\         x[1]         - 2.0 * x[3]  <= 2.0
  /\  -2.0 * x[1] + x[2] -       x[3]   <= 0.0
  ;

constraint
      y[1] +     y[2] - 2.0 * y[3] >= 2.0
  /\  y[1] +                 y[3]  >= 3.0
  /\       - 2.0*y[2] -      y[3]  >= -1.0
  ;
```

Un autre exemple ...suite

```
% ------------ PRIMAL --------------
solve maximize 2.0*x[1]+ 3.0*x[2]-x[3];
output ["x (primal) =", show(x),
        ",\n objectif primal (max) = ", show(2.0*x[1]+ 3.0*x[2]-x[3])];


% ----- Test extrémum Primal -------
% mzn-g12mip primal-dual-poly.mzn
% Ou minizinc -b mip primal-dual-poly.mzn
% vecteur x (primal) =[1.0, 2.0, 0.0]
% objectif primal (max) = 8.0


% ------------ DUAL --------------
solve minimize 3.0*y[1]+ 2.0*y[2];
output ["y (dual) =", show(y),
    ",\n objectif dual (min) = ", show(3.0*y[1]+ 2.0*y[2])];

% mzn-g12mip primal-dual-poly.mzn
% vecteur y (dual) =[2.6666666666666665, 0.0, 0.3333333333333326],
% objectif dual (min) = 8.0
```

➟ On constate le même extrémum .

Primal Dual en Minizinc

- Un autre exemple en Minizinc (remarquer les extrémums)
- Primal :

```
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
%x=[0.5, 1.25]                    objective=4.75
```

➡ $objective = 4.75$

Primal Dual en Minizinc ...suite

- Dual :

```
% La fome duale

set of int : 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)];

%mzn-g12mip dual.mzn
%y=[0.3125, 0.0, 0.25]                    objective=4.75
```

➟ $objective = 4.75$

Application Dual : un exemple de modélisation

Bernard est un producteur de fruits qui fabrique quatre cocktails de fruits ($C1, C2, C3, C4$) à partir des jus de fruit qu'il obtient de ses fruits selon les pourcentages du tableau suivant (le cout n'inclut pas le prix des fruits):

	jus Poire	jus Pomme	jus Abricot	cout (mélange+sucre+emballage)
C1	1/2	1/4	1/4	3
C2	1/2	1/2	0	2
C3	0	1/2	1/2	2,5
C4	1/3	1/3	1/3	4

La production de fruits et l'obtention de jus de chaque fruit coûte à Bernard (pour un litre de chaque jus, avant les mélanges):

Poire	Pomme	Abricot
2,5	1	2

Bernard vendrait en détails sur le marché ces cocktails aux prix de litre suivants :

C1	C2	C3	C4
6	4,5	5	6,5

➛ Mais Bernard souhaite plutôt trouver un repreneur (si possible) !

Application Dual : un exemple de modélisation ...suite

Un commerçant (Pierre) est intéressé par cette affaire et veut bien acheter les jus de fruit dont ces cocktails sont tirés.

Pierre propose d'acheter le stock de jus fruit de Bernard.
- Il doit lui proposer un prix qui empêchera la concurrence de Bernard.
- Un prix tel que Bernard ne continue pas à fabriquer ses propres cocktails.

- Le producteur a en stock 20 litres de Poire, 20 de Pomme et 15 d'Abricot.

1) Proposer à Bernard une manière de maximiser ces bénéfices.
2) Quel (min) prix d'achat des jus bruts Pierre devrait proposer à Bernard ?

Application Dual : un exemple de modélisation ...suite

Forme Primale (simplifiée, voir Addendum pour les explications) :

```
var float : x1; var float : x2;var float : x3;var float : x4;    % Primal
var float : Benefs;
var float : y1; var float : y2; var float : y3;                  % Dual

%================== PRIMAL ====================
constraint % Primal
      x1 >= 0.0 /\ x2 >= 0.0 /\ x3 >= 0.0 /\ x4 >= 0.0
  /\   1.5*x1 + 1.5*x2            + x4        <= 60.0
  /\  0.75*x1 + 1.5*x2 + 1.5*x3   + x4        <= 60.0
  /\  0.75*x1           + 1.5*x3   + x4        <= 45.0
  /\  Benefs = x1+0.75*x2+ x3+0.675*x4
  ;

solve maximize Benefs;

%-------------------------------------------------
% Résolution :
% mzn-g12mip jus-de-fruit.mzn
% Ou minizinc -b mip jus-de-fruit2.mzn
```

Solution :

Bénéfice= 52.5, Couts = 247.5
Litres de chaque Cocktail = [30.0, 10.0, 15.0, 0.0]

Quantité utilisée de chaque jus (Poire,Pomme,Abricot) : 20.0 20.0 15.0

Application Dual : un exemple de modélisation ...suite

La forme Duale :

- La forme Primale et Duale :

$$
\begin{array}{lcl}
Max & Z = x_1 + 0.75x_2 + x_3 + 0.675x_4 & \\
s.t. & 1.5x_1 + 1.5x_2 + x_4 & \leq 60.0 \\
 & 0.75x_1 + 1.5x_2 + 1.5x_3 + x_4 & \leq 60.0 \\
 & 0.75x_1 + 1.5x_3 + x_4 & \leq 45.0 \\
 & x_i \geq 0, x_i \in \mathbb{R} &
\end{array}
$$

$$
\begin{array}{lcl}
Min & Z' = 60y_1 + 60y_2 + 45y_3 & \\
s.t. & 1.5y_1 + 0.75y_2 + 0.75y_3 & \leq 1 \\
 & 1.5y_1 + 1.5y_2 & \leq 0.75 \\
 & 1.5y_1 + 1.5y_3 & \leq 1 \\
 & y_1 + y_2 + y_3 & \leq 0.675 \\
 & y_i \geq 0, y_i \in \mathbb{R} &
\end{array}
$$

- Le code *Minizinc* de la partie Duale $\cdots \rightsquigarrow$

Application Dual : un exemple de modélisation ...suite

- **La formulation de la Duale :**

```
constraint       % Dual
        y1 >= 0.0 /\ y2 >= 0.0 /\ y3 >= 0.0
  /\    1.5*y1 + 0.75*y2 + 0.75*y3    >= 1.0
  /\    1.5*y1 + 1.5*y2               >= 0.75
  /\    1.5*y1            + 1.5*y3    >= 1.0
  /\        y1 +      y2 +       y3    >= 0.675
  ;

solve minimize 60.0*y1 + 60.0*y2+ 45.0 *y3;
output ["Objectif min (Dual) : " , show(60.0*y1 + 60.0*y2+ 45.0 *y3),
       " (+couts de base obtenu du Primal)", "\n"];

% mzn-g12mip jus-de-fruit2.mzn
% Objectif min (Dual) : 40.25 (+couts de base obtenu du Primal)
% --> Donc, sans vendre sur le marché, il faut donner 40,25 + 247.5
```

➙ On avait les bénéfices maximisés = 52,5 (avec un cout total de 247,5)

➙ La valeur 40.25 montre que les bénéfices de Bernard seront dans tous les cas $>= 40,25$

➙ Pierre devra donc donner au moins $40,25 + 247,5 = 287,75$ à Bernard s'il veut que ce dernier accepte de lui céder ses jus de fruits.

OR : IP et B & B

- En LP, la technique B & B est différent de CP (mais même but)
  - ➙ En LP, c'est une technique de résolution d'un système LP
  - ➙ En CP, c'est une technique d'optimisation globale (à partir d'une 1e solution)

☞ Il est possible d'appliquer cette technique de LP en CP (à la main)
Mais certains environnement CLP la proposent (cf. problèmes complexes).

- **B & B** (en LP) : choix d'une variable (*branch*) + contraintes moins strictes (relaxation) sur cette variable + un "cout" de référence déjà calculé (*bound*).

- Deux étapes en B & B : résoudre une version relaxée du problème initial + Découper le problème en sous problèmes .
  - ➛ Sur un noeud, relaxer quelques contraintes et résoudre une instance relaxée du problème

- Relaxation :
  - ➙ Considérer le problème initial à un noeud P
  - ➙ Relâcher quelques contraintes et générer P_{rel} plus facile que P $\cdots \rightsquigarrow$

OR : IP et B & B ...suite

- **Exemples de relaxation** :
  - Au lieu de minimiser une variable, lui affecter une *binf* (resp. maximisation : *bsup*)
  - Réécrire par exemple $X \in \mathbb{N} \in [0,1]$ en $0 \geq x \leq 1$
  - Etc.

- La relaxation d'un problème initial P (avec une solution Z) donne le problème P_{rel} (avec une solution Z_{rel})

  ➙ Cas de minimisation : pour la solution optimale Z^* du problème P, la solution optimale de P_{rel} : Z^*_{rel} où $Z^*_{rel} \leq Z^*$

  ➙ Au noeud P, si *binf* est plus grande que la meilleure solution trouvée (*bound*), on peut abandonner le sous arbre correspondant (*Fathoming*).

  ☞ Cette dernière technique correspond à B&B utilisée avec CP/CLP.

- Branching :
  ➙ Choisir une variable
  ➙ Imposer plus de contraintes sur la variable

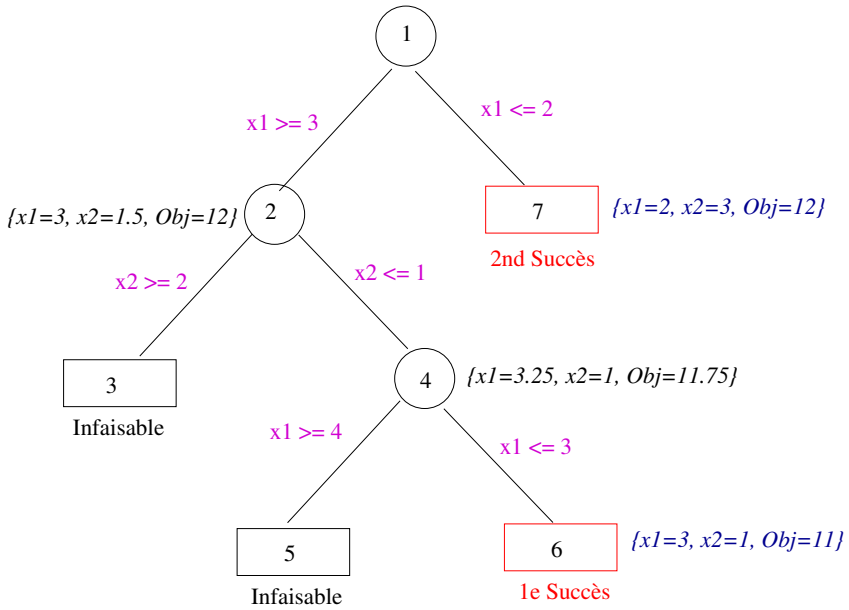
Détails avec un exemple de B & B

- *Le système IP originel :*

$$\begin{aligned} &\textit{Maximize } 3x_1 + 2x_2 \\ &\textit{Subject to} \\ &\qquad 4x_1 + 2x_2 \leq 15 \\ &\qquad x_1 + 2x_2 \leq 8 \\ &\qquad x_1 + x_2 \leq 5 \\ &\qquad x_i \geq 0, x_i \ \textit{Integer} \end{aligned}$$

- *La solution optimale (finale) :* $x_1 = 2, x_2 = 3, Obj = 12$
- *Voir ci-dessous pour les détails.*

- Exploration **"en profondeur d'abord"** (voir ordre des noeuds) :



- Lorsque qu'un système originel IP (ou MIP) ne donne pas une solution évidente respectant les contraintes d'intégralité, on a recours à une relaxation.

- Dans un premier temps, on ignore la contrainte d'intégralité et on résout le système LP.

  ➛ Pour notre exemple, on obtient $x_1 = 2.5, x_2 = 2.5, Obj = 12.5$. $\cdots \rightsquigarrow$

Détails avec un exemple de B & B ...suite

☞ Brancher sur une variable x consiste à envisager la partition binaire $x \leq \lfloor r \rfloor$ et $x \geq \lceil r \rceil$

➛ La constante r est obtenue par la résolution du système LP ci-dessus

➛ On choisit la variable x_1 avec $r = 2.5$ (on peut aussi bien commencer par x_2, voir +loin)

① *Brancher en noeud 1 (à gauche) sur* x_1 *avec* $x_1 \geq 3$ *correspond à envisager le système :*

Maximize $3x_1 + 2x_2$

Subject to

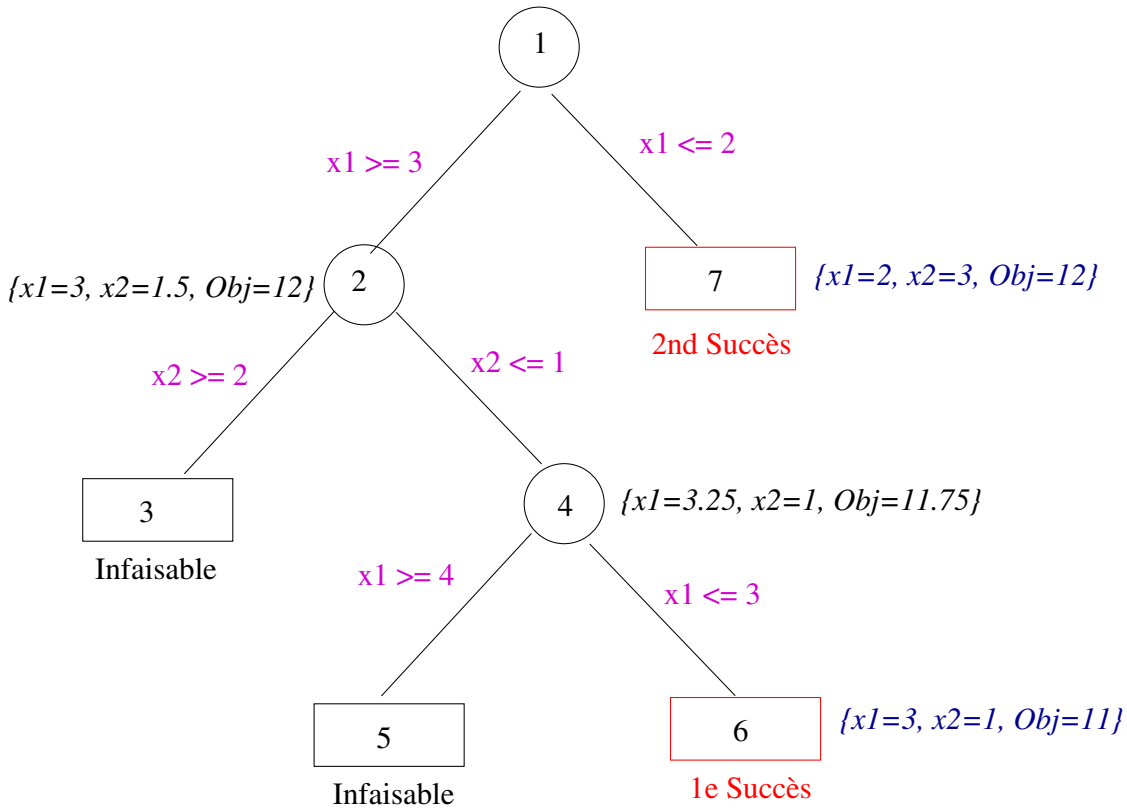
$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_1 \geq 3$$
$$x_i \geq 0, x_i \text{ Integer}$$

Et (à droite, examiné plus tard) sur x_1 *avec* $x_1 \leq 2$ *correspond au système :*

Maximize $3x_1 + 2x_2$

Subject to

$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_1 \leq 2$$
$$x_i \geq 0, x_i \text{ Integer}$$



Détails avec un exemple de B & B ...suite

② *On obtient la réponse* $\{x_1 = 3, x_2 = 1.5, Obj = 12\}$

Ce résultat n'est pas complet ($x_2 \in \mathbb{R}$) *: on branche sur* x_2 *pour laquelle* $r = 1.5$ *(Ici,* r *est fixée par la réponse ci-dessus).*

➛ *Le nouveaux systèmes (branchement sur* x_2*) :*

- *A gauche :* $x_2 \geq 2$ *(la contrainte sur* x_1 *est conservée)*

Maximize $3x_1 + 2x_2$

Subject to

$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_1 \geq 3$$
$$x_2 \geq 2$$
$$x_i \geq 0, x_i \text{ Integer}$$

- *Et à droite :* $x_2 \leq 1$ *(la contrainte sur* x_1 *est conservée)*

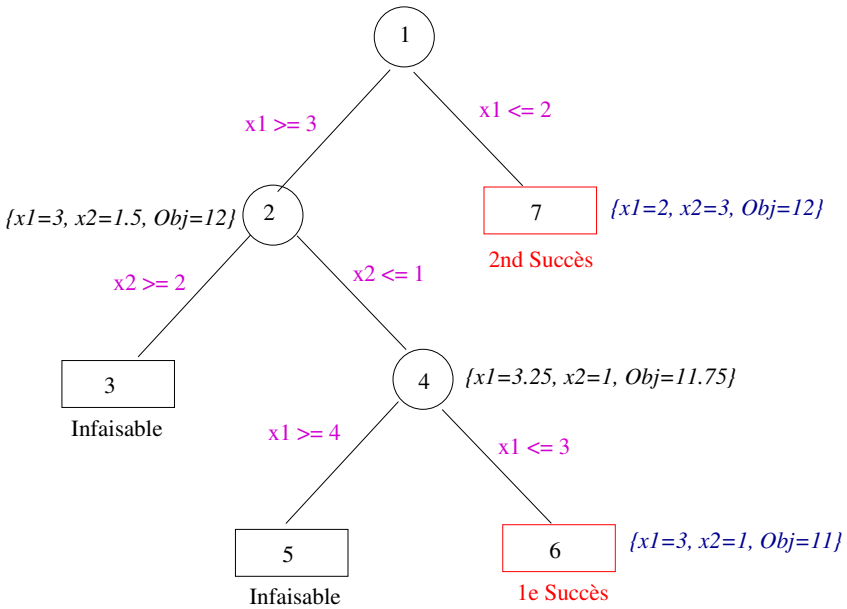
Maximize $3x_1 + 2x_2$

Subject to

$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_1 \geq 3$$
$$x_2 \leq 1$$
$$x_i \geq 0, x_i \text{ Integer}$$

☞ Notons qu'on pourrait décider de développer la branche droite du noeud auquel cas nous serions dans un parcours **"en largeur d'abord"**.

➛ On décide de continuer en profondeur d'abord.



Détails avec un exemple de B & B ...suite

③ *En noeud 3, le système (avec $x_2 \geq 2$) n'a pas de solution.*

On considère donc la branche droite du noeud 2 (avec $x2 \leq 1$).

④ *La réponse obtenue :* $\{x_1 = 3.25, x_2 = 1, Obj = 11.75\}$

➛ *Mais on a encore* $x_1 \in \mathbb{R}$.

On branche donc sur x_1 *avec* $r = 3.25$

➛ *Le nouveaux systèmes (branchement sur* x_1*) :*

- *A gauche :* $x_1 \geq 4$ *(la contrainte sur* x_2 *est conservée)*

Maximize $3x_1 + 2x_2$

Subject to

$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_1 \geq 3, x_2 \leq 1$$
$$x_1 \geq 4 \qquad x_i \geq 0, x_i \text{ Integer}$$

- *Et à droite :* $x_1 \leq 3$ *(la contrainte sur* x_2 *est conservée)*

Maximize $3x_1 + 2x_2$

Subject to

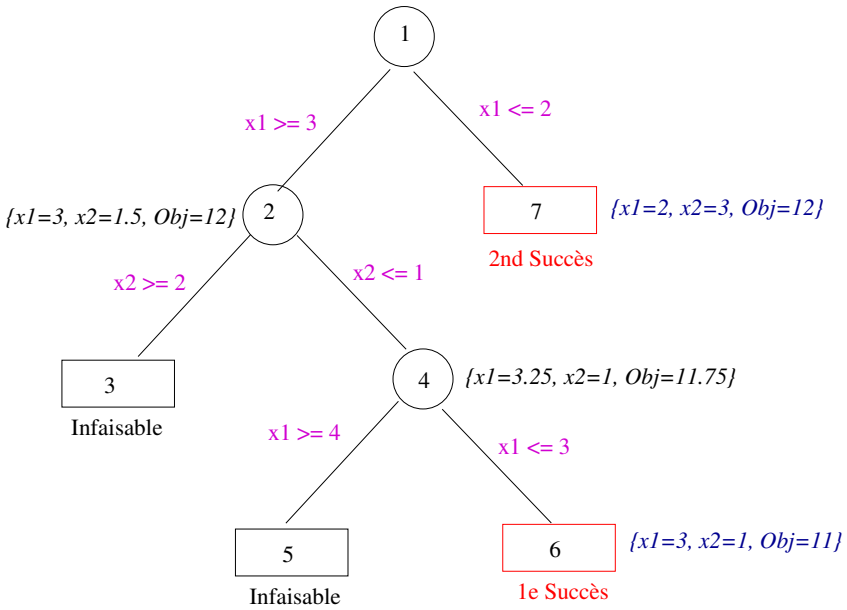
$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_1 \geq 3, x_2 \leq 1$$
$$x_1 \leq 3 \qquad x_i \geq 0, x_i \text{ Integer}$$

⑤ En noeud 5, le système (avec $x_1 \geq 4$) n'a pas de solution.

⑥ En noeud 6, le système donne la solution $\{x_1 = 3, x_2 = 1, Obj = 11\}$

Puisque les contraintes d'intégralité sont respectées, **le noeud 6 est un succès** .

☞ Il faut développer les branches en attente.



Détails avec un exemple de B & B ...suite

① *On revient au noeud 1 pour développer la branche droite (avec* $x_1 \leq 2$*)*

Le système LP à résoudre était :

Maximize $3x_1 + 2x_2$

Subject to

$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_1 \leq 2$$
$$x_i \geq 0,\ x_i \text{ Integer}$$

⑦ *En noeud 7, le système LP donne la solution*

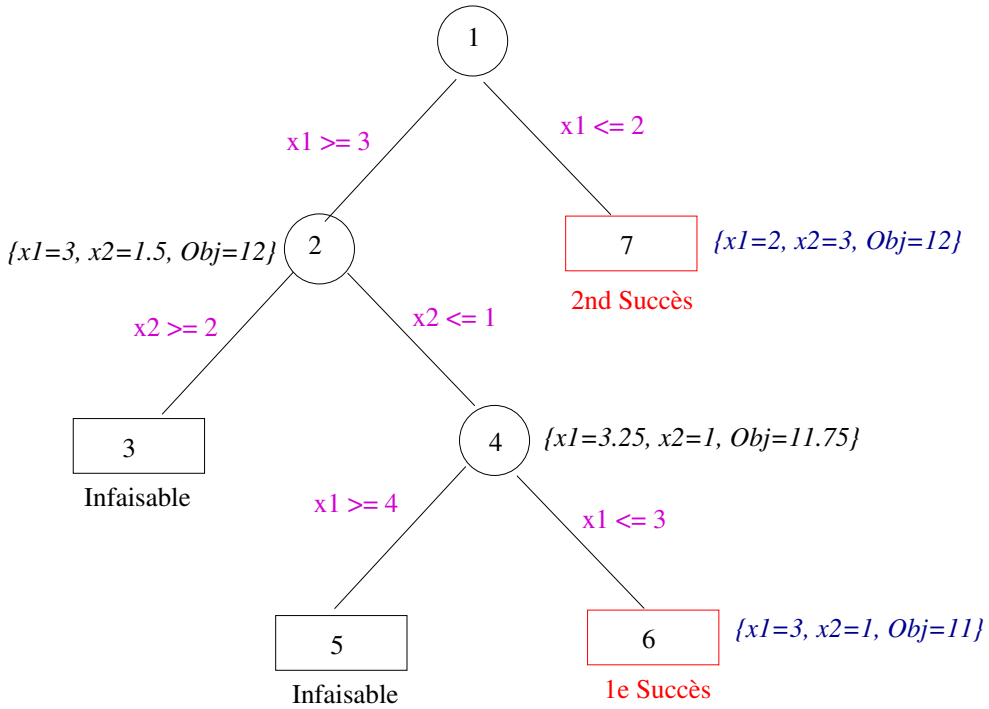
$$\{x_1 = 2, x_2 = 3, Obj = 12\}$$

Puisque les contraintes d'intégralité sont respectées, **le noeud 7 est un succès**.

Et puisque $Obj = 12$ *est mieux maximisé, on conserve plutôt cette solution.*

De plus, il n'y a plus besoin de brancher sur x_2 *(car* $x_1, x_2 \in \mathbb{N}$*).*

Le traitement est terminé car aucune branche n'est laissée en suspend.



Détails avec un exemple de B & B ...suite

☞ Il semble que l'ordre de développement ci-dessus était un ordre défavorable.

① *Si on avait développé d'abord la branche droite du noeud 1 avec le système :*

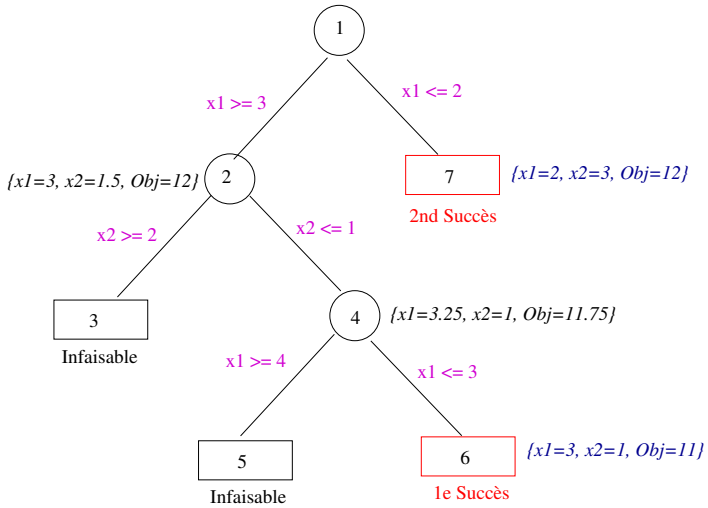
Maximize $3x_1 + 2x_2$

Subject to

$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_1 \leq 2$$
$$x_i \geq 0,\ x_i \ \textit{Integer}$$

On aurait d'abord obtenu la réponse $\{x_1 = 2, x_2 = 3, Obj = 12\}$.

Ensuite, en développant la branche gauche du noeud 1, on aurait trouvé sur le noeud 2 la réponse $\{x_1 = 3, x_2 = 1.5, Obj = 12\}$ ...



.. mais nous avions déjà la même valeur $Obj = 12$ tout en respectant les contraintes d'intégralité

➛ On réalise qu'il n'y a aucune chance d'améliorer Obj en développant le noeud 2 en branchant sur x_2 car nous ajouterions davantage de contraintes et (donc) la valeur d'Obj ne sera pas meilleure que 12.

➛ On aurait immédiatement arrêté le développement de l'arbre.

☞ Ceci correspond à **"bound"** (dans *branch-and-bound*) : on conserve la meilleure valeur d'Obj trouvée précédemment permettant de ne plus procéder à la relaxation et à la résolution des systèmes LP dont les réponses seront inférieures (ou au mieux égales) à la meilleure solution trouvé jusqu'à présent car la relaxation LP fournit une **borne supérieure** à toute solution qu'on pourra trouver en branchant sur d'autres variables.

Détails avec un exemple de B & B ...suite

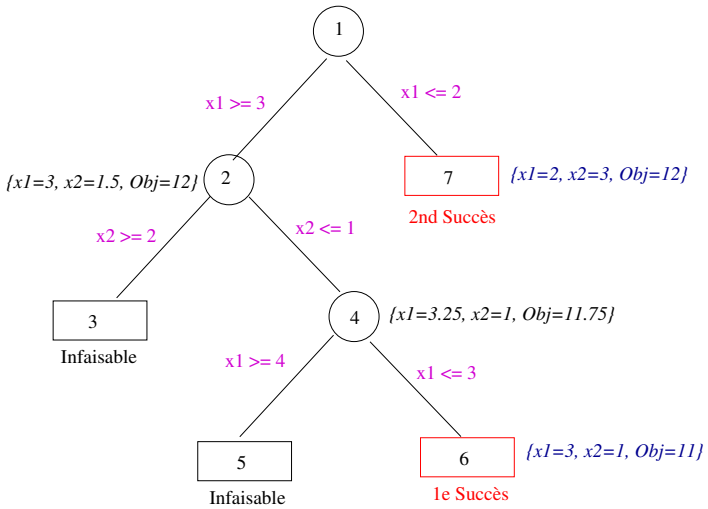
Interprétation géométrique de la solution :

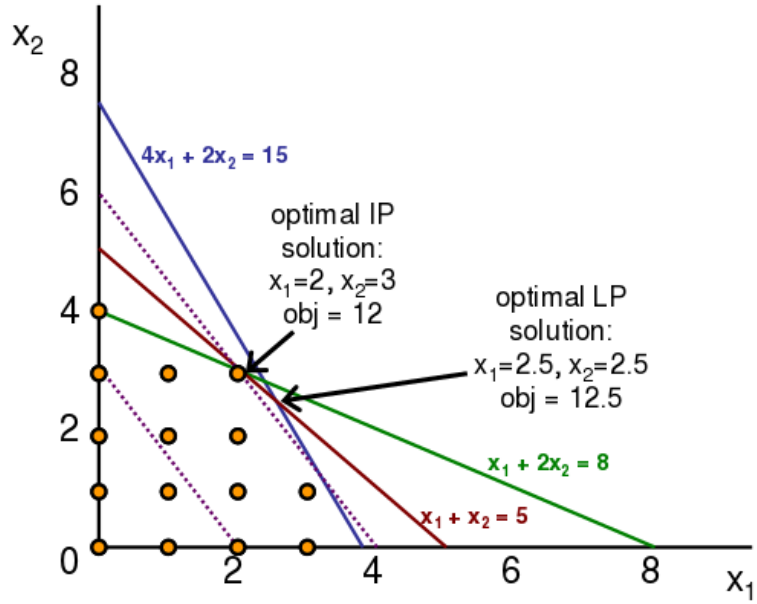
Le système originel à résoudre :

Maximize $3x_1 + 2x_2$

Subject to

$$4x_1 + 2x_2 \leq 15$$
$$x_1 + 2x_2 \leq 8$$
$$x_1 + x_2 \leq 5$$
$$x_i \geq 0,\ x_i \ \textit{Integer}$$





La solution optimale est $\{x_1 = 2, x_2 = 3, Obj = 12\}$.

La solution optimale au système LP (après relaxation) était $\{x_1 = 2.5, x_2 = 3.5, Obj = 12.5\}$

Les deux droites en pointillés correspondant à Obj (sens croissant de x_i) :

$$3x_1 + 2x_2 = 6$$
$$3x_1 + 2x_2 = 12$$

➛ la solution optimale est un point de la 2e équation.

Complément : un autre exemple B & B

La programmation entière (IP) est considérée comme une extension (ou un cas particulier) de LP.

Exemple IP :

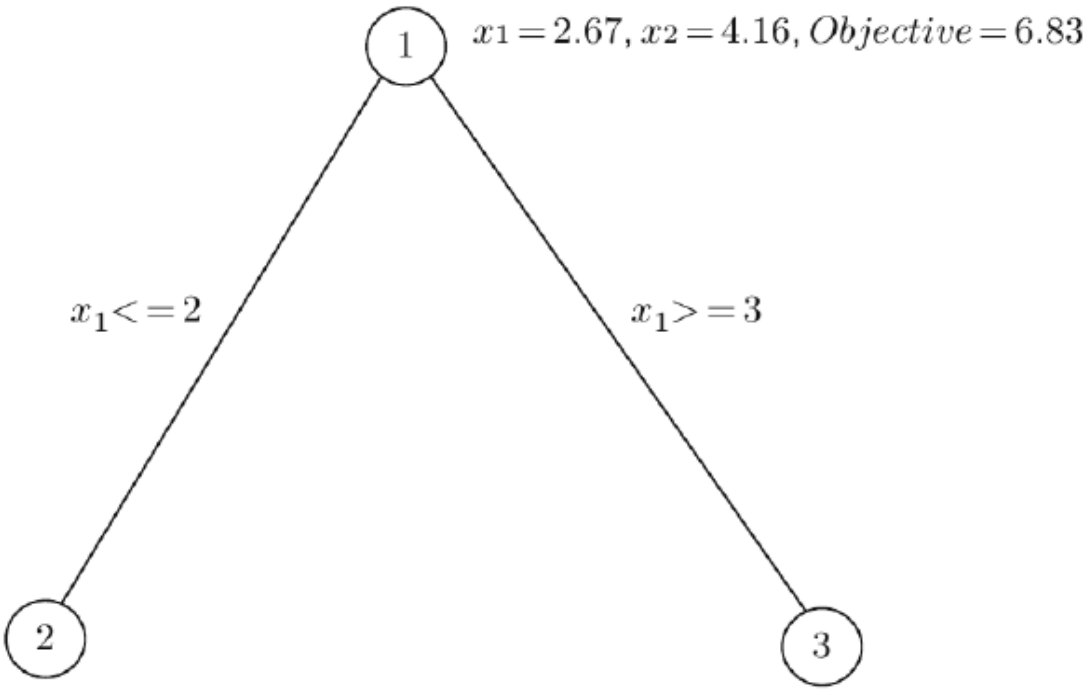
$$
\begin{aligned}
maximize \quad & Z = x1 + x2 \\
s.t. \quad & 2x1 + 2x2 \geq 3 \\
& -2x1 + 2x2 \leq 3 \\
& 4x1 + 2x2 \leq 19 \\
& x1, x2 \geq 0 \quad x1, x2 \in \mathbb{Z}
\end{aligned}
$$

Une solution au système relaxé (devenu LP) est ($x1, x2 \in \mathbb{Z}$ relaxées) :

$$
\begin{aligned}
x1 &= 2\frac{2}{3} \\
x2 &= 4\frac{1}{6} \\
Z &= 6\frac{5}{6}
\end{aligned}
$$

Complément : un autre exemple B & B ...suite

Branch and Bound : la relaxation de x_1 donne :

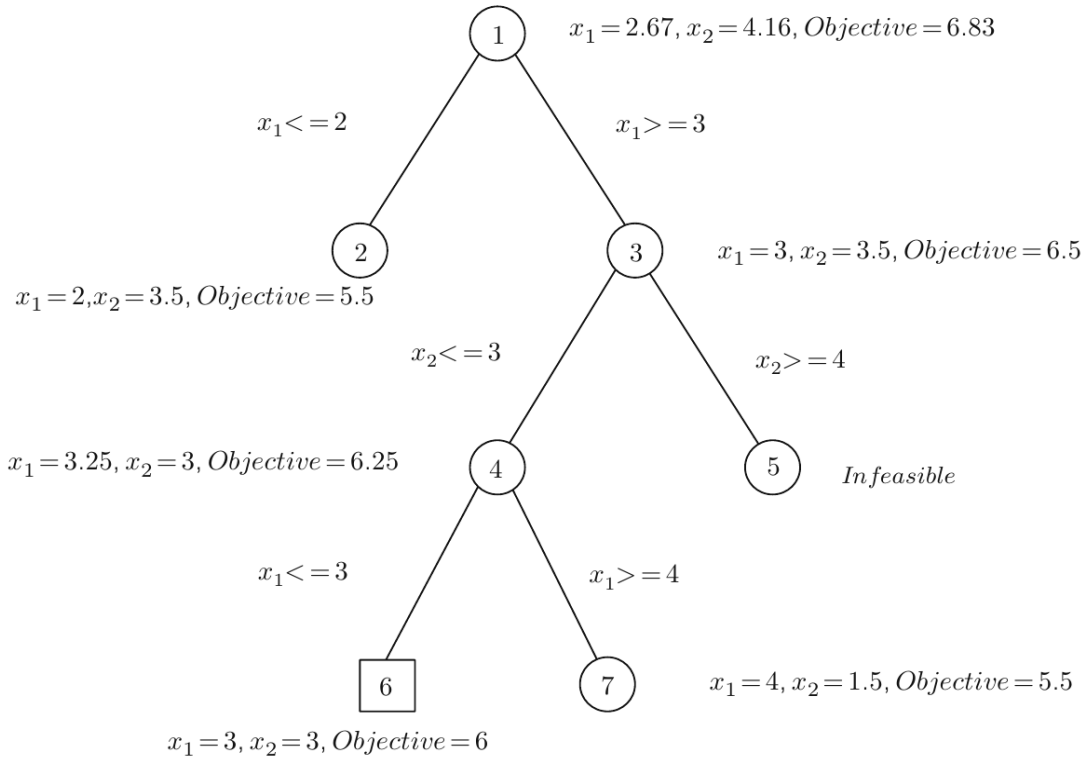


$x_1 = 2, x_2 = 3.5, Objective = 5.5$ $\quad$ $x_1 = 3, x_2 = 3.5, Objective = 6.5$

☞ On peut explorer les arbres (de relaxation / résolution) soit en profondeur-d'abord (Depth First) soit en largeur-d'abord (Breadth First).

Complément : un autre exemple B & B ...suite

L'arbre de relaxation complet (le noeud 6 est la solution) :



• Si on devait choisir de commencer à brancher sur $x2$ au lieu de x_1, l'arbre ne sera pas le même mais le résultat final sera le même.

• Une fois une contrainte imposée sur une branche (exprimée sur l'arête), cette contrainte s'appliquera à toutes branches qui en dérivent.

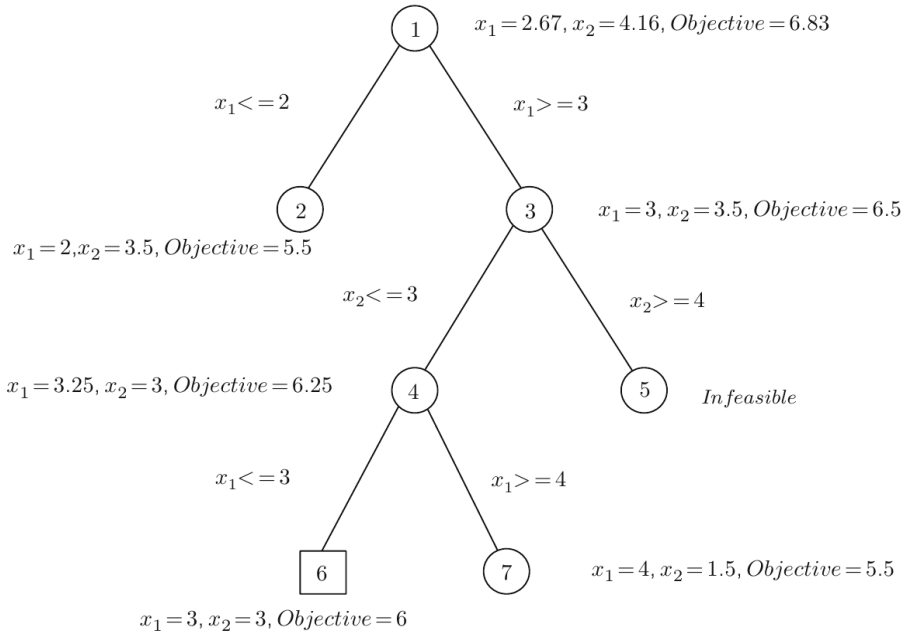
➙ Par exemple, la branche $x1 \geq 3$ dans tous les noeuds depuis le noeud 3.

Complément : Détails de l'exemple

Note : après un branchement, la variable choisie prendra la valeur de branchement (la valeur/contrainte sur l'arête, pas ce qui est obtenu sur un noeud) et ne déviera pas de cette contrainte dans la suite de l'arbre.

➙ C'est le cas par exemple des noeuds 3 et 4 ci-contre et la variable x_1 ($x_1 \geq 3$).

➙ Si une contrainte complémentaire est ajoutée sur une variable (cf. $x1 \geq 3$ sur 1-3 puis puis $x_1 \leq 3$ sur 4-6), les contraintes finales seront celles de l'intersection des contraintes :
➛ sur x_1 en 6 : $(x_1 \geq 3 \wedge x_1 \leq 3) \implies x_1 = 3$



• La méthode de fixation des variables **termine** même avec un nbr. exponentiel de branches (le nbr. de noeuds à chaque niveau se double avec 2^n noeuds au niveau n).

• Mais si une variable n'est pas limitée dans un domaine fini, alors la méthode B&B n'est pas garantie de terminer.

➛ De ce fait, B & B n'est pas formellement une méthode de décision pour IP.

☞ Voir le 2e exemple plus loin et un autre en Addendum.

Complément : Détails de l'exemple ...suite

Interprétation géométrique :

• $ABCD$ représente la région faisable du problème de relaxation de LP .

➙ La solution optimale de la relaxation LP correspond au point C.

• Le branchement sur $x1$ crée les deux régions $ABEF$ et GHD correspondant à la relaxation LP aux noeuds 2 et 3.
➙ Les points E et H correspondent aux solutions au niveau des ces noeuds.

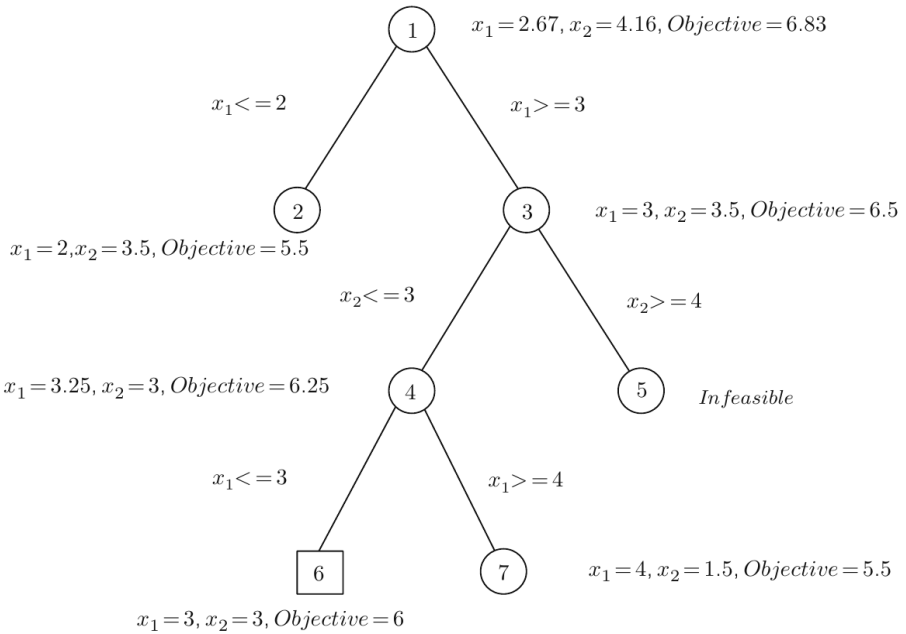
• Le branchement sur $x2$ (noeud 3) crée une région infaisable (région vide) pour $x2 \geq 4$ et la région $GIJD$ pour $x2 \leq 3$.

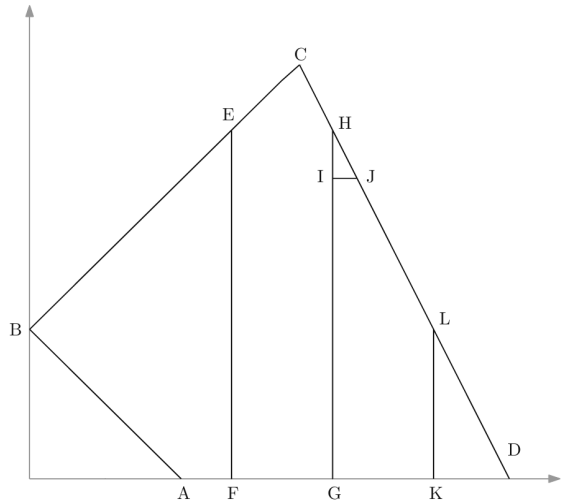
• La solution au niveau du noeud 4 correspond au point J.

• Le branchement sur $x1$ crée la région faisable GI à (une dimension) et KLD.

• Les solutions au niveau des noeuds 6 et 7 correspondent aux points Iet L.

• I correspond à la solution optimale entière (IP).





Plans de coupe

• **Cutting planes (plans de coupe)** est une alternative à B & B (pour IP).
  ◦ Résoudre une relaxation (P_{rel}) linéaire de P.
  ◦ Ajouter des plans (géométriques) de coupe quand la solution optimale de la relaxation n'est pas un (nombre) entier.

• Proche de *Simplex* : des droites (ou plans) viennent délimiter un peu plus la région des solutions, en particulier les solutions non entières.

• **L'idée de base :** considérer, à la place d'un IP, sa relaxation linéaire LP (sans intégralité) et d'exacerber ses inégalités en ajoutant d'autres inégalités acceptées par IP jusqu'à ce que (idéalement) une solution entière soit trouvée.

• Un plan de coupe est une **inégalité** supplémentaire admise par tous les points possibles de l'IP, mais pas par la solution LP actuelle (la relaxation).

• On réitère jusqu'à ce qu'une solution en nombre entier soit trouvée (sera automatiquement optimum pour IP) ou bien plus aucune inégalité est trouvée.

☞ Voir Addendum pour un exemple (et *Branch-and-Cut*) et "CP : Plan de coupe".

Génération de Colonnes

Génération de Colonnes :

• Utilisée pour les problèmes avec beaucoup de variables

• Ne pas considérer TOUTEs les variables mais seulement un sous ensemble de celles-ci (➙ donne *le Problème principal = Master problem*).

• Une fois le problème principal résolu, repêcher les variables délaissées qui améliorent la solution (donne des *Sous-Problèmes = Subproblems*).

N.B. : Cette technique a recours à la *théorie de la dualité* des variables (et de la réduction de cout) pour améliorer la solution.

• Il est parfois difficile de trouver les variables significatives

• N.B. : lien avec contrainte-réponse (en CP) et First-Fail ?

Remarques sur LP

• Pour une illustration de ces différentes techniques en LP, utiliser par exemple un outil tel que GLPK.

➙ GLPK produit des traces de la relaxation, de la tentative de résolution de la forme Duale, etc.

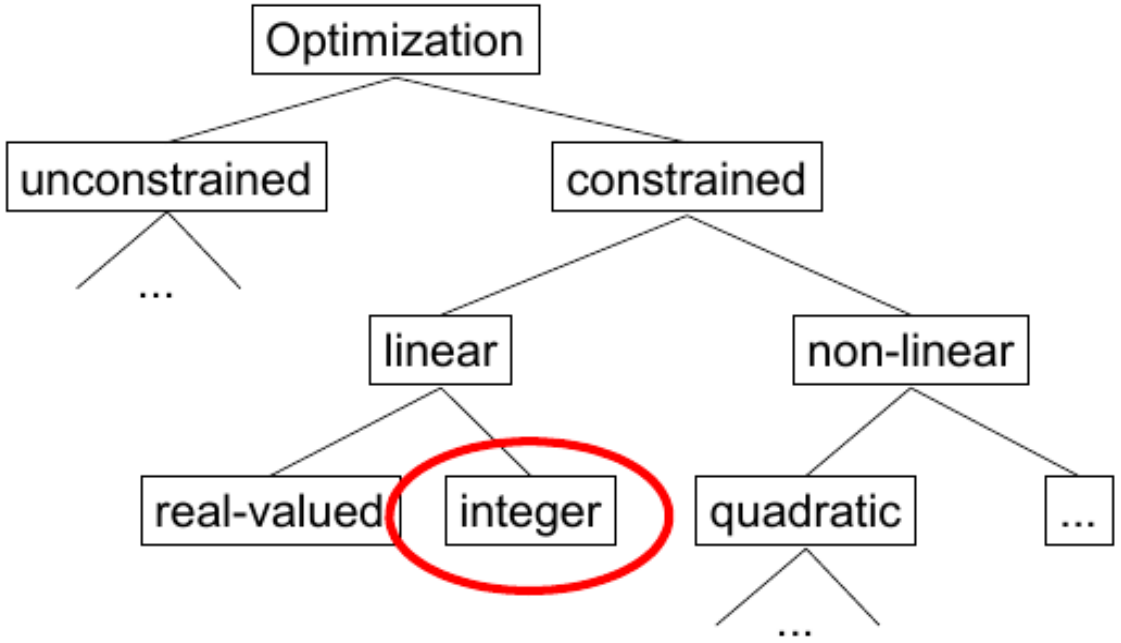
Qu'est que la CP ?

Un domaine relativement récent (CSP + IA) avec Applications en :

- Chaîne de montage (*Job shop* scheduling)
- Affectation fréquence téléphones cellulaires
- Planification (divers domaines : hôpitaux, maintenances, construction, etc)
- Planification portes d'embarquement aéroports, équipages aériens
- Industrie (planification, stock, etc)
- Planification (éducation)
- Nucléaire (Planification)
- Transports (tournés, personnel, etc)
- Planification production
  chimiques, aviation, raffinage du pétrole, acier, bois de charpente, plateaux photographiques, pneumatique, ...
- Etc.

Éléments de CP/CSP

- Si on est perdu !



- Format général : $min/max(\hat{x}) \quad s.t. \quad C(\hat{x})$
- On peut aussi travailler dans $\mathbb{R}$ (MIP)

☞ On n'est pas toujours obligé d'avoir une fonction objective.

Éléments de CP/CSP ...suite

Un exemple simple : problème de conversion C-F

- **Modélisation** : on sait que
  - Il s'agit d'une relation linéaire : $F = M * C + B$
  - 212 degrés F° = 100 degré C° : $212 = M * 100 + B$
  - 32 degrés F° = 0 degré C° : $32 = M * 0 + B$

- Donc : $\{F = M * C, 212 = M * 100 + B, 32 = M * 0 + B\}$

  ➛ On obtient : $\{-1.8 * C + F = 0.0, B = 32.0, M = 1.8\}$ (Contrainte réponse)

- Version avec des *nombres rationnels :*
  $\{F = M * C, 212 = M * 100 + B, 32 = M * 0 + B\}$;

  ➛ On obtient : $F = (9/5)C, M = 9/5, B = 32$

- Domaines : Nombres : entiers, relatifs, réels (représentation par intervalles), Bool, Rationnels, Symboliques, Chaînes, Ensembles, ...

☞ **A propos de la résolution :** *X in R* pour CSP/CP/CLP d'une part et *Simplex* et la méthode *Primal-Dual intérieur point* pour (N)LP

A propos de X in R

Un exemple :

Traduction en contraintes plus basiques (schéma *X in R*) :

2*X + 3*Y + 2 < Z, est traduit en contraintes plus simples.

- On connaît le domaine de X :: $min(X)..max(X)$, idem pour Y et Z.
- Réduction des bornes des domaines et / ou des valeurs des domaines :

```
'X=Y+C'(X,Y,C) :-      % LookAhead partiel
    X in min(Y)+C .. max(Y)+C,  Y in min(X)-C .. max(X)-C.
```

Exemple : si dom(Y)= {1,3,4,7,9} et C=2, alors X peut prendre une valeur dans 3..12 (extrémums)

```
'X=Y+C'(X,Y,C) :-          % LookAhead complet
    X in dom(Y)+C ..Y in dom(X)-C.
```

Exemple : si dom(Y)= {1,3,4,7,9} et C=2, alors X peut prendre une valeur dans {3,5,6,9,11}

Exemples

Exemple2 : Prêt $\in \mathbb{R}$

$$versement(Capital, Mois, Taux, Due, Mensualite) : -$$
$$\{Mois = 1, Due = Capital + (Capital * Taux - Mensualite)\}.$$

$$versement(Capital, Mois, Taux, Due, Mensualite) : -$$
$$\{Mois > 1, Mois1 = Mois - 1,$$
$$Capital1 = Capital * (1 + Taux) - Mensualite$$
$$\},$$
$$versement(Capital1, Mois1, Taux, Due, Mensualite).$$

Questions :

- Quelle mensualité pour un capital = 100 sur 12 mois, taux=10% (on ne doit rien à la fin)
  $versement(100, 12, 0.1, 0, M).M = 14.67$
- Combien emprunter et quelle est la mensualité sur 12 mois à un taux annuel de 10% pour que l'on doive capital /2 à la fin.
  $versement(C, 12, 0.1, C * 0.5, M).M = 0.12 * C$

  ➙ Pour transformer cette **contrainte-réponse** en une réponse (on précise C ou M) :
  - Et si on rembourse 15 euros par mois ?
    ➙ $versement(C, 12, 0.1, C * 0.5, 15). \rightarrow C = 121.57$
  - Et si on emprunte 150 euros ?
    ➙ $versement(150, 12, 0.1, C * 0.5, M). \rightarrow M = 18.5$
  - Combien emprunter pour une mensualité = 15 (sur 12 mois, taux=10%, reste = 0)
    ➙ $versement(P, 12, 0.1, 0, 15). \rightarrow P = 102.20$

Exemples ...suite

Exemple 3 : on reprend l'exemple *Stones* pour une résolution en CP.

- Une pierre de 40 kg se brise en 4 morceaux.
- Avec ces morceaux, on peut peser tous les objets de 1 a 40 kg sur une balance à 2 plateaux.
- Quel est le poids de chacun des 4 morceaux ?

Modélisation :

- 4 variables (entiers) pour les 4 morceaux (soit L=$[P_1, P_2, P_3, P_4]$)
  - ➙ $P_i \in 1..40, \quad \sum P_i = 40.$
- Les 4 poids devraient être différents (plus pratique !) : $P_i \neq P_j, i \neq j$.
- Tous les poids 1..40 sont mesurables par les 4 morceaux.

- On regroupe l'ensemble des contraintes :

$P1, P2, P3, P4 :: 1..40,$
$P_1 + P_2 + P_3 + P_4 = 40,$
$P1 \leq P2, P2 \leq P3, P3 \leq P4,$ % pour éviter la symétrie
$tous\_les\_poids\_mesurables([P1, P2, P3, P4], 1..40)$

Exemples ...suite

tous_les_poids_mesurables$(L, 1..N)$:

Pour $X = 1..40$, vérifier que X est mesurable par $P1, P2, P3, P4$.

- Pour chaque mesure, on envisage les deux plateaux de la balance.

  ➙ On associe B_i à P_i, $B_i \in -1.. + 1$ avec :

  - $B_i = -1$: P_i sur le plateau de gauche
  - $B_i = +1$: P_i sur le plateau de droite
  - $B_i = 0$: P_i ne participe pas pour mesurer le poids X

- $\forall X \in 1..40$, *peser* X *avec* $P1, P2, P3, P4$: $X = \sum_{i=1..4} B_i P_i$

  $X = B_1 * P_1 + B_2 * P_2 + B_3 * P_3 + B_4 * P_4$,
  $B_i :: -1..1$.

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

Exemples ...suite

Modélisation CLP :

- Pour simplifier, on n'utilise pas la matrice :
  - ➙ On ne donne pas la solution "comment peser K kg".
  - ➙ Appels récursifs de la "relation" *tous_les_kilos_pesables*

- Tous les poids entiers entre A et B doivent être mesurables (*pesables*).

```
tous_les_kilos_pesables(A_peser, Jusque, Liste_pierres) :-
    A_peser < Jusque,                    % pas encore fini.
    peser(A_peser, Liste_pierres),  % peser 'A_peser' kg  avec 4 pierres
    Suivant is A_peser +1 ,
    tous_les_kilos_pesables(Suivant, Jusque, Liste_pierres).


peser(A, [P1, P2, P3, P4]) :-   % Pour peser un poids donné A :
    % Choisir les B_i tels que les 4 pierres disposés de part
    L = [B1,B2,B3,B4],
    % et d'autre de la balance fassent A kg.
    A = B1*P1 + B2*P2 + B3*P3 + B4*P4,
    %  B_i prend sa valeur dans -1 (à gche), 0 (absence) ou 1
    L :: [-1,0,1].
```

Exemples ...suite

- Rappel de la solution Minizinc

```
include "increasing.mzn";   % pour casser la symétrie

par int : masse = 40;       % masse est un "par"amètre (vs. var)
par int : nombre_pierres = 4;

set of int : a_peser = 1..masse;

array[1..nombre_pierres] of var 1..masse : pierres;
array[a_peser,1..nombre_pierres] of var -1..1 : combinaisons;

constraint
    % On brise la symétrie du problème
    increasing(pierres)
  /\
    % La somme des masses des pierres est egale a la masse totale recherchee
    sum(p in 1..nombre_pierres)(pierres[p]) = masse
  /\
    % Chacune des masses a peser est obtenue par addition
    %    ou soustraction des masses de pierres
    forall(m in a_peser)(
        sum(p in 1..nombre_pierres)(pierres[p]*combinaisons[m,p]) ==m
    )
  ;

solve satisfy;
output[show(pierres),"\n",show(combinaisons)];
```

Exemples ...suite

- Test :

```
% minizinc stone.mzn
%   [1, 3, 9, 27]
%   [1, 0, 0, 0,
    -1, 1, 0, 0,    % pour peser 2kg : la pierre de 3kg d'un côté, de 1kg de l'autre
     0, 1, 0, 0,
     1, 1, 0, 0,
    -1,-1, 1, 0,    % pour peser 5kg : 9kg d'un côté, 1kg et 3kg de l'autre
     0,-1, 1, 0,
     1,-1, 1, 0,
    -1, 0, 1, 0,
     0, 0, 1, 0,
     1, 0, 1, 0,    % pour peser 10kg : 9kg et 1kg d'un même côté

    -1, 1, 1, 0,      0, 1, 1, 0,      1, 1, 1, 0,      -1,-1, -1, 1,
     0,-1,-1, 1,      1,-1,-1, 1,     -1, 0,-1, 1,       0, 0, -1, 1,
     1, 0,-1, 1,     -1, 1,-1, 1,      0, 1,-1, 1,       1, 1, -1, 1,
    -1,-1, 0, 1,      0,-1, 0, 1,      1,-1, 0, 1,      -1, 0,  0, 1,
     0, 0, 0, 1,      1, 0, 0, 1,     -1, 1, 0, 1,       0, 1,  0, 1,
     1, 1, 0, 1,     -1,-1, 1, 1,      0,-1, 1, 1,       1, -1, 1, 1,
    -1, 0, 1, 1,      0, 0, 1, 1,      1, 0, 1, 1,      -1,  1, 1, 1,
     0, 1, 1, 1,
     1, 1, 1, 1     %  40 Kilos : toutes les pièrres utilisées
    ]
```

➙ La matrice combinaisons contient 40 lignes et 4 colonnes (dans -1..1).

Éléments de la CSP / CP

- CSP permet une spécification déclarative des problèmes ;
  ➙ La spécification séparée des techniques de résolution
- Le programmeur est libéré des considérations opératoires ;
  ➙ il se concentre sur la définition des relations et les contraintes
- Le système de programmation s'occupe de la résolution dirigée par les contraintes.
- Il n'y a plus de système de résolution spécifique à inventer pour chaque problème .
- Le programmeur formalise le problème dans le langage CSP en suivant les schémas de spécification.
- Dans un environnement CLP, on dispose en plus d'un démonstrateur de théorèmes.

Éléments de la CSP / CP ...suite

- Les étapes de spécification d'une solution :
  - ➛ absence des considération de codage
  - ◦ Faire un découpage hiérarchique et trouver une définition par contraintes;
  - ◦ Spécifier Variables, Domaines et les relations;
  - ◦ Procéder éventuellement à une adaptation à l'outil ;
  - ◦ Spécifier les générations de valeurs;
  - ◦ Vérifier (éventuellement) les propriétés des réponses ;
  - ◦ Optimiser éventuellement.
  - ◦ Maîtrise des méthodes, niveau requis

- Choix d'une méthode selon la complexité $|D|^{|V|}$

  ➛ Exemple $n - reines$

Éléments de la CSP / CP ...suite

Rappel : Comme on a vu plus haut

"libéré de considérations opératoires" veut par exemple dire d'utiliser les contraintes de plus haut niveau sans devoir les traduire.

- Rappel de quelques exemples :
  - $f(x) \leq O \ \vee \ g(x) \leq 0$
  - $f(x) > 0 \implies g(x) \geq 0$ on sait que $A \implies B \equiv \neg A \vee B$
  - $X \neq Y$
  - $x, y \in 1..n, x \neq y$
  - *constraint alldifferent(tab);*
  - $element(i, [a_1, a_2, ..., a_n], x)$ qui veut dire : $\exists i, x = a_i$
- Les variables 0..1 permettent des modélisations intéressantes.
- Même si *Minizinc* (p. ex.) dispose des contraintes de haut niveau, on peut avoir parfois recours à ces techniques d'optimisation.

Éléments de la CSP / CP ...suite

Le rôle des contraintes dans un CSP

- Une contrainte représente une relation entre les objets
- Une contrainte peut spécifier :
  - Une information partielle / incomplète :
    ➙ l'age du capitaine est au moins 40 ans.
  - Une information floue :
    ➙ l'age du capitaine est environ 40 ans (intervalle)
- Une contrainte est déclarative : indépendante du processus de calcul
- Une contrainte n'est pas orientée, elle spécifie une relation :
  ➙ Dans X + Y = Z , si deux des 3 variables sont connues,
  la 3e est calculée.
- L'ordre de l'expression des contraintes n'influence pas sur le sens du programme.

Éléments de la CSP / CP ...suite

- En CSP, le domaine de calcul doit être connu :
  - Dans $X^2 = 2$

    Si le domaine est $\mathbb{N}$ ➙ pas de solution

    Si le domaine est $\mathbb{R}$ ➙ deux solutions
  - Dans $X^2 - Y^2 = 0, X^2 + Y^2 = 2$.

    le domaine de X et Y ne doivent pas changer dans la conjonction.

Aperçu de la complexité des problèmes Csp

Exemples de combinatoire :

1. Crypt-arithmétique ($\{S, E, N, D, M, O, R, Y\} \in 0..9$)
   S E N D + M O R E = M O N E Y
   e.g. 9 5 6 7 + 1 0 8 5 = 1 0 6 5 2
   ➛ 108 combinaisons

2. Composition de mots croisés (pour une grille ordinaire et un lexique de 150 mots équitablement répartis)
   ➛ 3! 4! 11! 17! 15! 30! 33! 26! 5! solutions potentielles

3. Planification de la rotation de 10 bateaux dans 5 emplacements d'un port comporte env. $5^{10} \sim 10^6$ possibilités :
   ➛ 3 minutes de calcul pour un ordinateur testant une solution par ms.
   Mais, pour 20 bateaux et 10 emplacements (10^{20} combinaisons)
   ➛ plus de 50 millions d'années sur le même ordinateur !

- Choix de modélisation selon $|D|^{|V|}$

Aperçu de la complexité des problèmes Csp ...suite

- Un cas pas si rare que cela : Sudoku

De combien de manières peut-on remplir une grille Sudoku ?

➙ 6,670,903,752,021,072,936,960

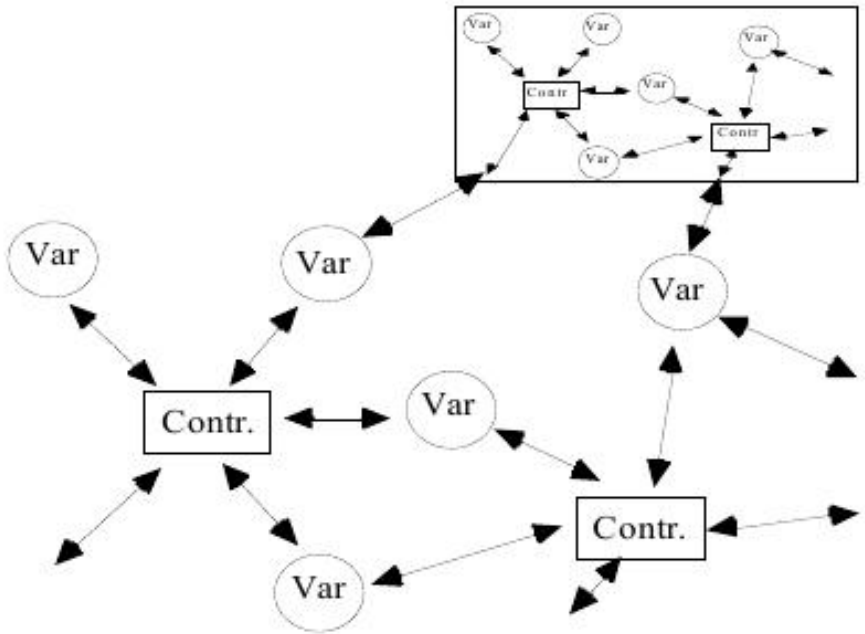
5	9	3	7	6	2	8	1	4
2	6	8	4	3	1	5	7	9
7	1	4	9	8	5	2	3	6
3	2	6	8	5	9	1	4	7
1	8	7	3	2	4	9	6	5
4	5	9	1	7	6	3	2	8
9	4	2	6	1	8	7	5	3
8	3	5	2	4	7	6	9	1
6	7	1	5	9	3	4	8	2

Aperçu de la complexité des problèmes Csp ...suite

Graphe des contraintes et Structure d'un CP (V, D, C)

- Définition des variables (V)
- Expression des contraintes (D et C)
- Énumération éventuelle pour la recherche d'une/des solutions optimales.

➙ Graphe des contraintes



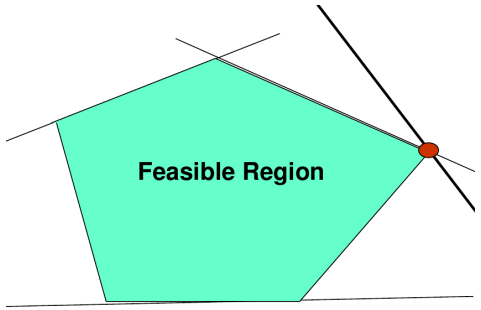
Aperçu de la complexité des problèmes Csp ...suite

Notion de Contrainte-réponse (en CLP)

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

• Les sortie sont appelées contraintes-réponses

➙ Une contrainte-réponse est une évaluation partielle du programme : une instance plus spécifique du programme.



☞ Une *Contrainte-réponse* délimite une région de solutions (*faisables*)

☞ La notion de *Contrainte-réponse* es très liée à la Propagation ...↝

Propagation : PERT

Rappel de l'exemple d'ordonnancement (PERT) :

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

- Les contraintes de précédence / différence :

avant(T1, T2). noté $T1 < T2$
avant(T1, T3).
avant(T2, T6).
avant(T3, T5).
avant(T4, T5).
avant(T5, T6).

diff(T2, T3). % T2 et T3 disjonctives ($\neq$)

Propagation : PERT ...suite

Consistance et propagation des contraintes :

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

$T1 \in \{1,2\}$, $T2 \in \{2,3,4\}$, $T3 \in \{2,3\}$, $T4 \in \{1,2,3\}$, $T5 \in \{3,4\}$, $T6 \in \{4,5\}$

• On procède ensuite par énumération (les domaines sont finis)
Par exemple : $T1 = 2 \rightarrow T2 = 4$, $T3 = 3$, $T4 \in \{1,2,3\}$, $T5 = 4$, $T6 = 5$.

• Si l'on est contraint à terminer toutes les tâches en 4 heures :

➙ $Max(Ti) \leq 4$
➙ $T6 \in \{4,5\}$, $T6 = 4$
➙ $T5 < T6$, $T5 \in \{3,4\} \rightarrow T5 = 3$
➙ $T4 < T5$, $T4 \in \{1,2,3\}$, $T5 = 3 \rightarrow T4 \in \{1,2\}$
➙ $T3 < T5$, $T3 \in \{2,3\}$, $T5 = 3 \rightarrow T3 = 2$
➙ $T1 < T3$, $T1 \in \{1,2\}$, $T3 = 2 \rightarrow T1 = 1$
➙ $T2 \in \{2,3,4\}$, $T2 < T6$, $T6 = 4 \rightarrow T2 \in \{2,3\}$

Propagation : exemple fret

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

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

$$
\begin{aligned}
min \quad & 90x_1 + 60x_2 + 50x_3 + 40x_4 \\
s.t. \quad & 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 \\
& x_1 + x_2 + x_3 + x_4 \leq 8 \\
& x_i \in \{0, 1, 2, 3\} \qquad \text{maxi 3 dispo}
\end{aligned}
$$

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

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

Propagation : exemple fret ...suite

$$
\begin{aligned}
min \quad & 90x_1 + 60x_2 + 50x_3 + 40x_4 \\
s.t. \quad & 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 \\
& x_1 + x_2 + x_3 + x_4 \leq 8 \\
& x_i \in \{0, 1, 2, 3\}
\end{aligned}
$$

- Estimation de x_1 par propagation :

$$7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42$$

➙ $7x_1 \geq 42 - 5x_2 - 4x_3 - 3x_4$

Si x_2, x_3, x_4 devaient prendre leur valeur maximale ($U_j = 3$).

$$\boxed{x_1 \geq \lceil \frac{42 - 5 * 3 - 4 * 3 - 3 * 3}{7} \rceil = 1}$$

➙ Ce qui permet de réduire le domaine de x_1 : $x_1 \in \{1, 2, 3\}$

- CP procède à consistance des bornes (cf. ci-dessus : *Look-Ahead* Partiel)

Propagation : exemple fret ...suite

La consistance de bornes (*Arc-consistance faible*) :

- Soit $\{L_j, ..., U_j\}$ le domaine de x_j
- Un ensemble de contraintes est bornes-consistant si pour tout j :
  $x_j = L_j$ dans certaine solution faisable et
  $x_j = U_j$ dans certaine solution faisable.
- Le but de la Bornes-consistance :
  → Ne pas affecter à x_j à une valeur infaisable au cours des branchements.
  → Alléger le graphe des contraintes
- Bornes-consistance s'applique "bien" pour une seule inégalité.
  $7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42$
  est Bornes-consistant pour $x_1 \in \{1, 2, 3\}$ et $x_2, x_3, x_4 \in \{0, 1, 2, 3\}$.

  Mais pas nécessairement pour un ensemble d'inégalités.

☞ N.B. : en général, toute propagation est suivie d'un test de consistance,
➙ Cette interaction forte entre les deux mène parfois à une confusion !

Propagation : exemple fret ...suite

• Rappel : la propagation Bornes-consistance ne s'applique pas bien pour un ensemble d'inégalités.

• Par exemple, dans le LIP (BIP) :

$$x_1 + x_2 >= 1,$$
$$x_1 - x_2 >= 0,$$
$$x_1, x_2 \in \{0, 1\}$$

➟ Les solutions : $(x_1 = 1, x_2 = 0)$ et $(x_1 = 1, x_2 = 1)$

• Ici, on devrait pouvoir éliminer (par Bornes-consistance) 0 du domaine de x_1
➟ (cause de non succès d'élimination : 2 inégalités).

• La propagation Bornes-consistance n'a pas d'effet sur (l'intérieur) des domaines (ce sera *Arc-consistance forte* ou *Look-Ahead complet*).

Consistance : plus de détails

• Un ensemble de contraintes est consistant si toute affectation (assignation de valeur) partielle des variables qui ne viole aucune contrainte est faisable.

- ◦ peut être étendue à une solution faisable.
- ◦ cf. exemple PERT (ne veut pas dire que toute combinaison est ok)

• Par la consistance, toute affectation partielle infaisable est explicitement exclue par une contrainte.

• On peut résoudre un problème (algorithmiquement, CLP) avec un ensemble de contraintes consistants sans retour en arrière (BT).

➙ Mais il faut une Path-consistance (ou hyper ou k-consistance).

☞ La *k-consistance* permet davantage de souplesse :
Au lieu de tout ou un (arc), on peut fixer $k \geq 1$ (k : nbr d'arcs considérés)
➙ On peut faire par exemple *3-consistance*.

Consistance : plus de détails ...suite

Exemple :

$X_1 + X_{100} \geq 1$
$X_1 - X_{100} \geq 0$
$\ldots$ Autres contraintes sur $x_{i=1..100}$ (on a 100 variables)
$X_i \in \{0, 1\}$

Ce système n'est pas consistant car l'assignation $x_1 = 0$ (qui ne viole aucune contrainte) est pourtant infaisable

➙ C-à-d. : aucune solution ne contient $x_1 = 0$.

➙ On risque de développer un sous-arbre avec 2^{99} noeuds (autres 99 variables sauf x_1) sans jamais atteindre une solution (car $x_1 = 0$ dans ce sous-arbre).

➙ L'ajout de $x_1 = 1$ rend cet ensemble consistant (élimine le sous-arbre).

• Le but d'un algorithme k-consistance est de réaliser cette opération en excluant des valeurs des domaines des variables.

Consistance : plus de détails ...suite

Consistance de Noeud (Node Consistancy)

• C'est la forme la plus simple de la vérification de contraintes.

• Un noeud représenté par une variable V est **noeud-consistant** (dans le graphe des contraintes C) si toutes les valeurs du domaine de V vérifient les contraintes unaires (dont le nombre de variables =1, p. ex. $X > 1$) sur V.

➙ Les valeurs du domaine qui ne vérifient pas ces contraintes sont éliminées du domaine de V.

• Exemples : $X > 0, X > 10$ donnera $X > 10$
$X \geq 3, X \leq 3$ donne $X = 3$
$X > 3, X < 3$ est node-inconsistent.

Consistance : plus de détails ...suite

Consistance d'Arcs (Arc consistancy)

• Une fois la consistance de noeud vérifiée :
➙ on considère les contraintes binaires faisant participer 2 variables
→ (qui correspondent aux arcs dans le graphe des contraintes)

• La consistance d'arcs d'un réseau : consistance sur toutes paires X,Y (liées par un ensemble de **contraintes binaires** $C_{x,y}$).

• Attention à la *direction* sur les contraintes :
➙ la consistance de $C_{x,y}$ ne vaut pas forcément pour $C_{y,x}$
➙ Exemple : "x=y" vs. "x>y".

• Pour une 3e variable Z du réseau des contraintes reliée à y:
➙ on vérifie la consistance des contraintes $C_{y,z}$
→ (modifient éventuellement les domaines Dy et Dz),
➙ on doit revoir la consistance de $C_{x,y}$ car le domaine Dy a pu changer.

Consistance : plus de détails ...suite

• Suite à la consistance d'arcs, le réseau de contraintes restant peut ne pas avoir de solution .

• **Exemple** :

◦ Si le domaine des variables X et Y est réduit à une seule même valeur, la contrainte $X \neq Y$ conduit immédiatement à un échec.

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

➙ La vérification de la consistance d'arcs ne peut supprimer aucune valeur de ces domaines.

➙ Pourtant, le réseau restant n'a pas de solution (d'où l'intérêt de la consistance de chemin, voir ci-dessous).

• De ce fait, la consistance d'arc ne suffit pas à supprimer le besoin de retours arrières dans les CSP/CLP (car échec local est possible).

Consistance : plus de détails ...suite

• N.B. : on dira que qu'un solveur qui se contente de ces vérifications sans le mécanisme de retour arrière est **incomplet**

➛ i.e. (ne donne pas toujours les bonnes réponses.

➛ Par contre, muni du mécanisme de retour arrière, ce solveur devient **complet**.

• La complexité de la procédure de consistance d'arc (algo. REVISE) : $O(b.k^2)$
- k borne la taille des domaines de valeurs des variables et
- b le nombre de contraintes binaires.

➛ La valeur k^2 est le nombre maximal de relations binaires dans le réseau de contraintes C.

• La complexité spatiale de cet algorithme est la même que sa complexité temporelle.

../..

Consistance : plus de détails ...suite

Consistance complète d'arcs (ou Hyper Consistance d'arc) :

• La consistance (partielle) d'arc modifie seulement les bornes des domaines.
➟ cf. *Bornes-consistance* (ou *Look-Ahead* partiel) plus haut.

• La consistance totale (Full Arc Consistancy) considère la totalité du domaine de chaque variable (mais nécessite des domaines bornés)
➟ Procède à davantage de propagations que la version partielle.
➟ *Look-Ahead* complet.

• Un ensemble de contraintes est *hyper-arc consistant* si chaque valeur dans chaque domaine de variable fait partie de quelque solution réalisable.
➟ Autrement dit : les domaines sont réduits autant que possible.

• Si toutes les contraintes sont binaires (contenant 2 variables), l'hyper-arc consistance = arc consistant.

• La réduction de domaine est le moteur principal de CP.

Consistance : plus de détails ...suite

A noter (CLP)

- Question d'efficacité.
- La consistance (simple) d'arcs suivie d'énumération (après l'énumération pour une variable, le réseau de contraintes élimine les valeurs incompatibles des domaines des autres)
- Ce qui permet de minimiser le nombre de retours arrières
- Cette technique est implantée dans une version optimisée dite *intelligent back-tracking*.

Consistance : plus de détails ...suite

Consistance de Chemin (Path consistancy) ou la K-consistance

• La consistance d'arc ne supprime pas forcément toutes les valeurs incompatibles

➟ ne suffit pas à supprimer les retours arrières dans le solveur.

• On peut étendre l'algorithme de consistance d'arc à la consistance du chemin (plusieurs variables en lien via des contraintes).

• Un graphe de contraintes est dit **k-consistant** si :

→ La consistance vérifiée pour k-1 variables du graphe ET

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

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

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

Consistance : plus de détails ...suite

- **Exemple** :
  Si le domaine $D = \{1, 2\}$ pour X, Y, Z
  avec les contraintes $X \neq Y, X \neq Z, Z \neq Y$ (2 à deux différentes) pour les variables,
  Alors l'algorithme conclue sur un échec immédiat.

- Coût important des algorithmes de consistance de chemin :
  ➙ on se contente en général d'une vérification de 3-consistance.

- Pour les contraintes binaires non orientées :
  ➙ algorithme *restricted path-consistancy*

☞ Important : si un graphe de contraintes sur N variables est N-consistant (path-consistant), alors aucune recherche (aucun retour arrière) n'est nécessaire pour trouver une solution.
➙ Par contre, même si le graphe est K-consistant pour $K < N$, alors on pourrait avoir besoin de procéder à des retours arrières.

Illustration de la Consistance

Exemple coloration (re-visité) :

- Soit à une étape donnée l'état de la coloration [Hooker-2007].

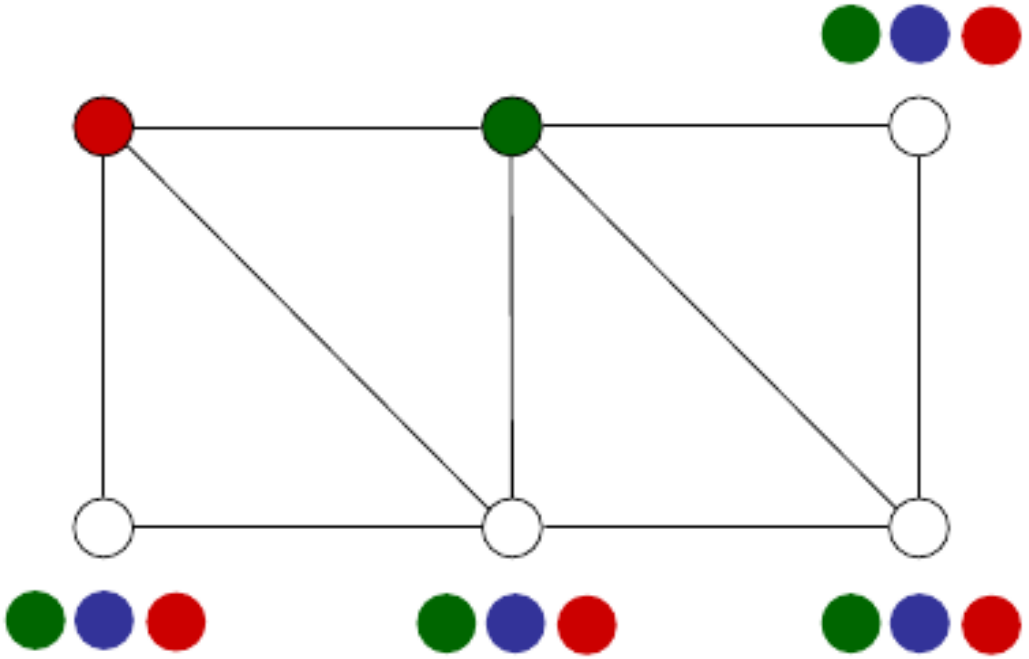


Illustration de la Consistance ...suite

- Les étapes de vérification de consistance et de propagation ($g \rightarrow d$, $h \downarrow b$)

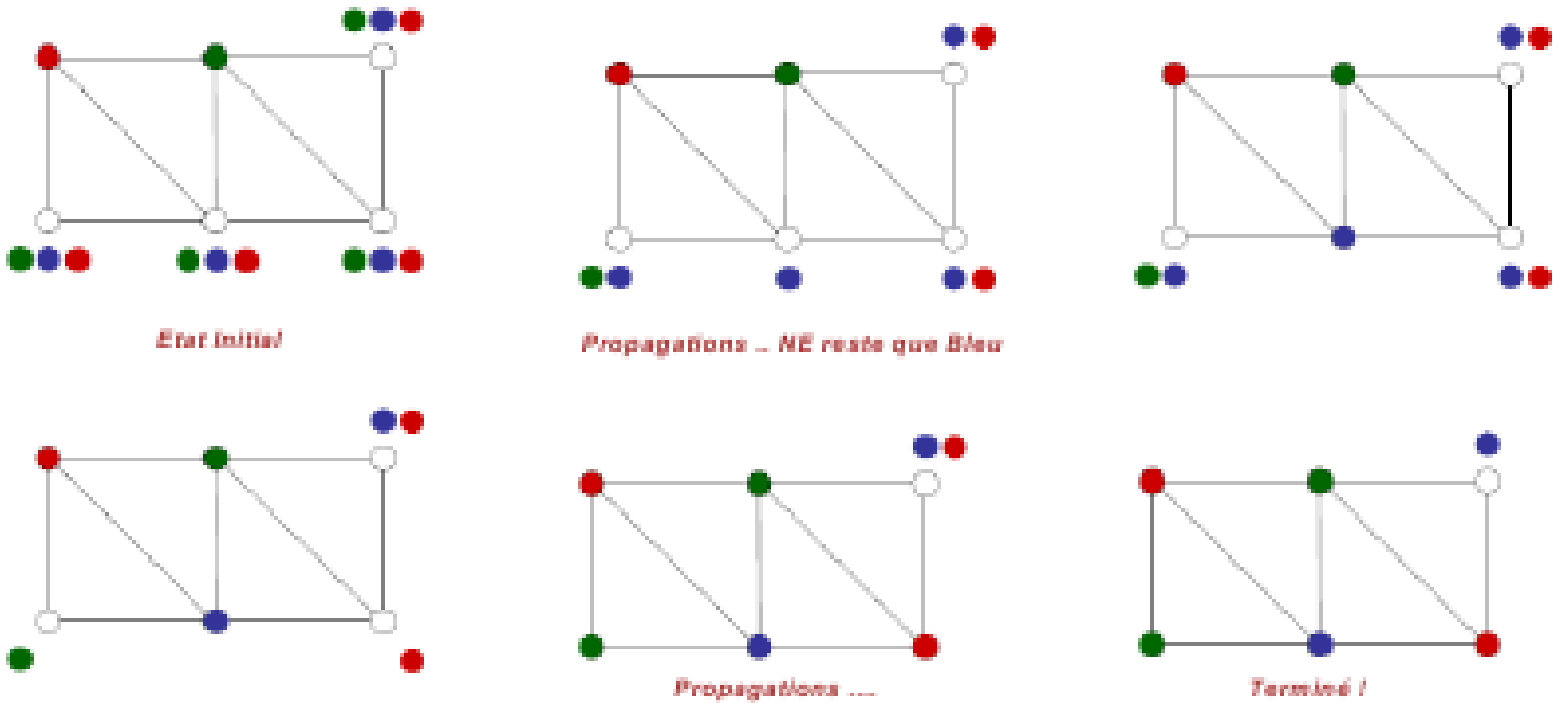


FIGURE 1 – Résolution par hyper-propagation

CP : Relaxation

Le même principe que pour LP est appliqué.

- Suite de l'exemple *Fret*, après Bornes-consistance (sur x_1).

  Rappel LP après Bornes-consistance :

$$\begin{aligned} min \quad & 90x_1 + 60x_2 + 50x_3 + 40x_4 \\ s.t. \quad & 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 \\ & x_1 + x_2 + x_3 + x_4 \leq 8 \\ & x_1 \in \{1,2,3\}, x_2, x_3, x_4 \in \{0,1,2,3\} \end{aligned}$$

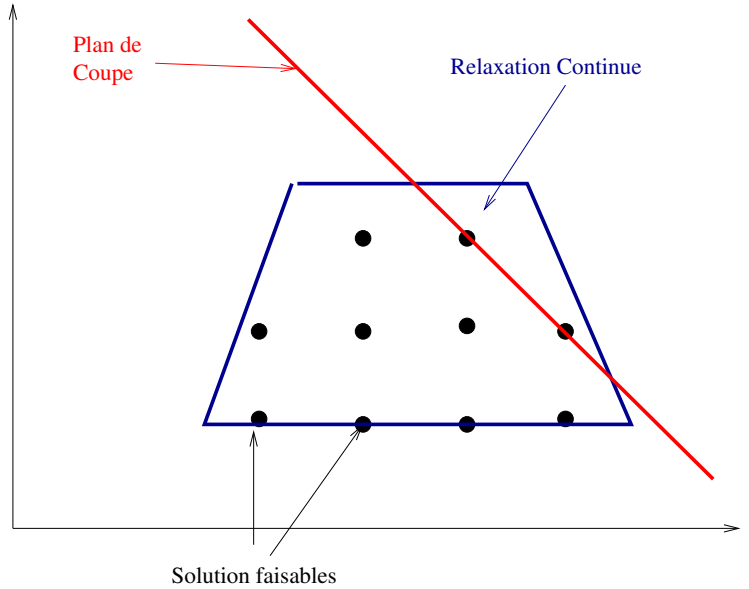
- On applique une **relaxation continue** :
  ➙ remplacement des domaines par les bornes :
  ➙ $x_1 \geq 1, \quad 0 \leq x_2, x_3, x_4 \leq 3$

- Le résultat est plus simple à résoudre que le LIP original.
  ➙ La valeur optimale de ce LP est une binf du LIP original.

CP : Plans de coupe

On reprend l'exemple :

$$
\begin{array}{ll}
min & 90x_1 + 60x_2 + 50x_3 + 40x_4 \\
s.t. & 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 \\
& x_1 + x_2 + x_3 + x_4 \leq 8 \\
& x_1 \geq 1, \quad 0 \leq x_2, x_3, x_4 \leq 3
\end{array}
$$



Plan de coupe :

• On peut créer une relaxation plus serrée (un minimum plus large) qui sera un **plan de coupe**.

• Toutes les solutions du problème originel satisferont (le doivent) un **plan de coupe valide**.

☞ Un plan de coupe peut exclure (on dit "**cut off**") certaines solutions de la relaxation continue.

CP : Plans de coupe ...suite

Comment trouver les plans de coupe ?

$$
\begin{array}{ll}
min & 90x_1 + 60x_2 + 50x_3 + 40x_4 \\
s.t. & \mathbf{7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42} \\
& x_1 + x_2 + x_3 + x_4 \leq 8 \\
& x_1 \geq 1, \quad 0 \leq x_2, x_3, x_4 \leq 3
\end{array}
$$

- Dans $7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42$,
  ➙ On remarque que $7x_1 + 5x_2$ seul ne peut pas satisfaire l'inégalité, même avec les Bsup de x_1, x_2 (qui sont $U_1 = U_2 = 3$).

$$\boxed{x_3 + x_4 \geq \lceil \frac{42 - (7 * 3 + 5 * 3)}{max\{x_4, x_3\}} \rceil = 2}$$

- Un plan de coupe (est trouvé) :
  On dit que les variables x_1 et x_2 constituent un packing (noté $\{x_1, x_2\}$) et le plan (la droite) $\mathbf{x_3 + x_4 \geq 2}$ une coupe sac-à-dos .

CP : Plans de coupe ...suite

Plan de coup (= inégalité valide)

- Soit x_i avec son domaine $[L_i, U_i]$.
- Soit la constante $a \geq 0$
- Un **packing** P pour $ax \geq a_0$ satisfait

$$\sum_{i \notin P} a_i x_i \geq a_0 - \sum_{i \in P} a_i U_i$$

Et génère une **coupe sac-à-dos**

$$\sum_{i \notin P} x_i \geq \lceil \frac{a_0 - \sum_{i \in P} a_i U_i}{\max\limits_{i \notin P} \{a_i\}} \rceil$$

- Cf. exemple plus haut.

☞ Tout plan de coup (valide) est *de facto* une "aide" pour le solveur.

CP : Plans de coupe ...suite

Résumons les plans de coupes possibles :

$$
\begin{aligned}
min \quad & 90x_1 + 60x_2 + 50x_3 + 40x_4 \\
s.t. \quad & \mathbf{7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42} \\
& x_1 + x_2 + x_3 + x_4 \leq 8 \\
& x_1 \geq 1, \quad 0 \leq x_2, x_3, x_4 \leq 3
\end{aligned}
$$

Et

Packing maxima	coupes sac-à-dos
$\{x_1, x_2\}$	$x_3 + x_4 \geq 2$
$\{x_1, x_3\}$	$x_2 + x_4 \geq 2$
$\{x_1, x_4\}$	$x_2 + x_3 \geq 3$

➙ Les autres "packings" sont en général non-maxima et / ou redondants

☞ Si possible, on les vérifie.

CP : Plans de coupe ...suite

• On a donc :

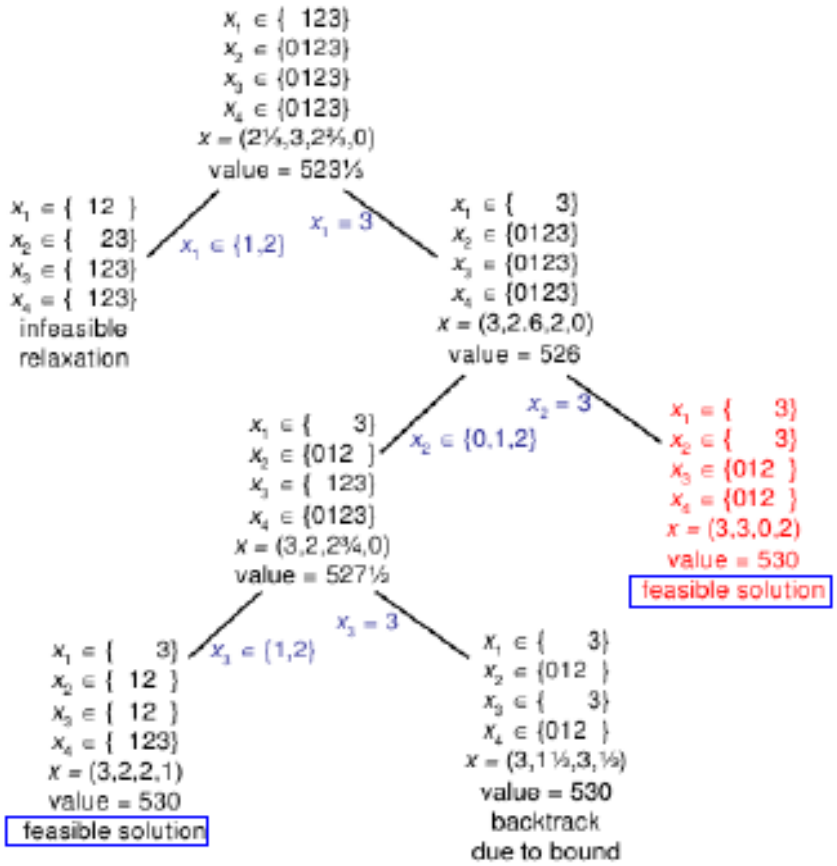
$$
\begin{aligned}
min \quad & 90x_1 + 60x_2 + 50x_3 + 40x_4 \\
s.t. \quad & 7x_1 + 5x_2 + 4x_3 + 3x_4 \geq 42 \\
& x_1 + x_2 + x_3 + x_4 \leq 8 \\
& x_1 \geq 1, \quad 0 \leq x_2, x_3, x_4 \leq 3 \\
& x_3 + x_4 \geq 2 \\
& x_2 + x_4 \geq 2 \quad \not\subset \text{ 3 coupes sac-à-dos} \\
& x_2 + x_3 \geq 3
\end{aligned}
$$

• La valeur optimale (523.3) de ce LP est une Binf de la valeur optimale du LIP originel (dont le minimum est 530 pour $x_1 = 3, x_2 = 2, x_3 = 2, x_4 = 1$).

• Pour résoudre ce système, on peut appliquer des techniques telles que *Branch-Infer-and-relax* (équivalente à branch and bound + relaxation de MP).

CP : Plans de coupe ...suite

- Par exemple (2 solutions faisables avec $X = (3, 2, 2, 1)$ et $X = (3, 3, 0, 2)$):



Méta-Stratégies

Stratégie appliquées en CP (peuvent être appliquées partout !)

- Soit les variables $X_1 \cdots X_n$, les domaines $D_1 \cdots D_n$.
  ➟ Un domaine $D_i = \{d_1, \cdots d_m\}$
- Les stratégies **générales** (méta stratégies)
  - Les méthodes et techniques qui regardent en arrière :
    - Générer-Tester (le plus inefficace) :
      ➟ Ne considère pas les contraintes lors d'affectation de valeurs individuelles aux variables.
      ➟ Choisit $X_n = d_n \in D_n$ tel que $\{X_1, ..., X_n\}$ satisfasse les contraintes.
      ➟ le test est fait trop tard !
    - Retour arrière (BackTrack) : une sorte de node consistancy
      ➟ Restreint le choix de $d_{k+1} \in D_{k+1}$ pour la variable X_{k+1} (suivant les contraintes)
      ➟ On essaie une valeur pour X_{k+1} en vérifiant les contraintes avec les valeurs (actuelles) de $X_1..X_k$.

Méta-Stratégies ...suite

- Les techniques qui regardent en avant :
  - **Forward Checking (BT + Arc consistancy)** appelé aussi *Look-Ahead Partiel*.

    ➙ (En plus de BT), le choix de d_{k+1} pour X_{k+1} laisse une chance à $X_{k+2} \cdots X_n$ (par node consistancy)

    ➙ Dans une forme partielle, on anticipe seulement sur X_{k+2}.

    ➙ Les vérifications sont faites entre la dernière variable instanciée et les restantes.

    ➙ Revient à tester si le domaine de $X_{k+2} \cdots X_n$ ne sont pas vides

    $\rightarrow' X = Y + C'(X, Y, C) : X \in min(Y) + C \cdots max(Y) + C,$
    $Y \in min(X) - C \cdots max(X) - C.$ % Look-Ahead partiel

Méta-Stratégies ...suite

- **Look Ahead (BT + Complete Arc consistancy)**

  ➙ En plus de (FC), on vérifie la satisfiabilité d'une solution possible pour les autres variables (deux à deux).

  ➙ C-à-d (Maximise l'anticipation de l'échec) :

  ➙ on vérifie qu'il y aura non seulement une chance pour chaque variable $X_{k+1} \cdots X_n$ sachant les valeurs de $X_1 \cdots X_k$,

  ➙ mais aussi qu'en plus, $X_{k+1} \cdots X_n$ se laissent deux à deux une chance possible et satisfaisante.

  ➙ Path-consistance pour $X_1..X_k$ et arc-consistance pour $X_{k+1} \cdots X_n$.

  → $'X = Y + C'(X, Y, C) : X \in dom(Y) + C..Y \in dom(X) - C.$

  % LookAhead complet

Méta-Stratégies ...suite

- Les stratégies FC et LA peuvent être mises en place
  ➛ Pour optimiser ou pour soulager le solveur (s'il fait de la consistance)

- Il y a quelques variantes de Retour Arrière (cf. *intelligent BT* , etc.)
- Dans sa forme basique, *BT* est utilisé avec *générer-tester*.
- La différence est dans le moment des vérifications (tests).
- Les stratégies telles que la Programmation Dynamiques, *B&B*, etc.

sont des formes plus évoluées de BT (avec des tests effectués à divers moments).

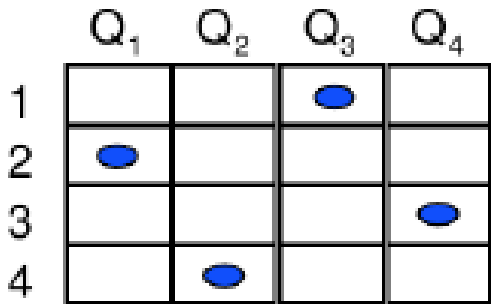
- **Rappel** : la solution complète = path consistancy + BT.

Un exemple

Illustration par les N-reines

- Énoncé :

  Placer 4 reines tel qu'elles ne s'attaquent pas (sur une ligne, colonne et diagonale), Soit Q_i = le numéro de ligne d'une reine dans la colonne i, $1 \leq i \leq 4$



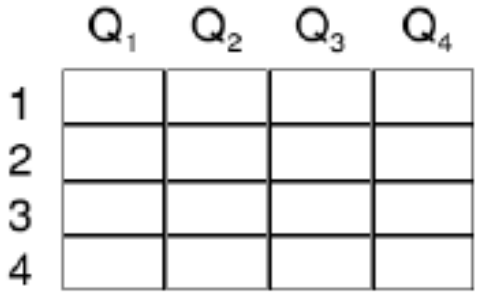
une solution à 4-rienes

Un exemple ...suite

- On exprime les contraintes *in extenso* pour la clarté :

$Q_1, Q_2, Q_3, Q_4 \in \{1, 2, 3, 4\}$
$Q_1 \neq Q_2$, $Q_1 \neq Q_3$, $Q_1 \neq Q_4$,
$Q_2 \neq Q_3$, $Q_2 \neq Q_4$,
$Q_3 \neq Q_4$,

$Q_1 \neq Q_2 - 1$, $Q_1 \neq Q_2 + 1$, $Q_1 \neq Q_3 - 2$, $Q_1 \neq Q_3 + 2$,
$Q_1 \neq Q_4 - 3$, $Q_1 \neq Q_4 + 3$,
$Q_2 \neq Q_3 - 1$, $Q_2 \neq Q_3 + 1$, $Q_2 \neq Q_4 - 2$, $Q_2 \neq Q_4 + 2$,
$Q_3 \neq Q_4 - 1$, $Q_3 \neq Q_4 + 1$



Un exemple ...suite

Méthode Générer-tester :

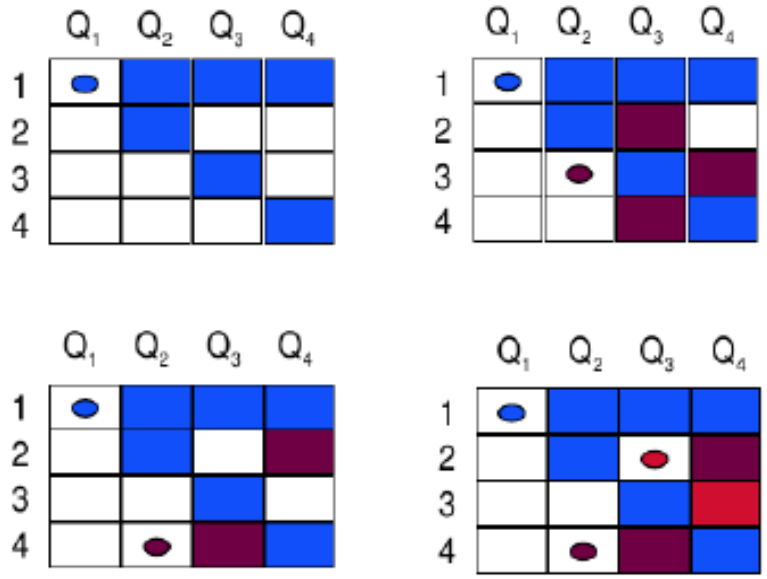
- Au total, 256 évaluations (= 44) :
  - ➙ 64 échecs avec Q_1=1 (4 x 4 x 4 = 64 possibilités pour Q_2, Q_3 et Q_4)
  - ➙ Echecs constatés (avec Q_1=1)
  - ➙ 48 échecs avec Q_1=2, 1 =< Q_2 =< 3, ... (Q_2=4 sera un bon placement)
  - ➙ 3 avec Q_1=2, Q_2=4, Q_3=1
  - ➙ un total de 115 évaluations pour trouver la première solution.

- **Méthode Retour arrière** : ...

Un exemple ...suite

- **Méthode Forward Checking (FC)**

une couleur par Qi. Un pion par colonne : simplifie une des contraintes.



De gauche à droite et d haut vers le bas : Q1=1 permet d'éliminer les cases bleues (et laisse {3, 4} à Q2)

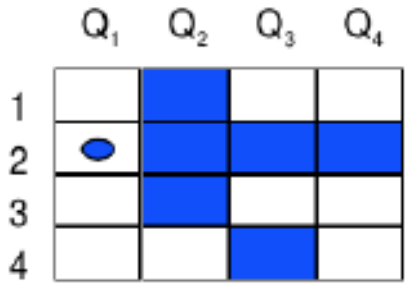
puis Q2=3 élimine les cases bordeaux (ne laisse rien à Q3); on défait Q2=3

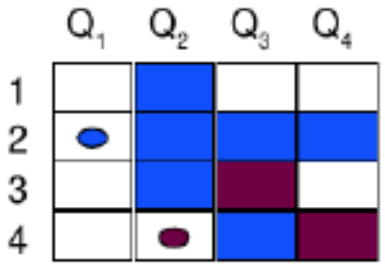
puis Q2=4 et Q3=2 ne laisse pas de chance à Q4. On défait Q1=1

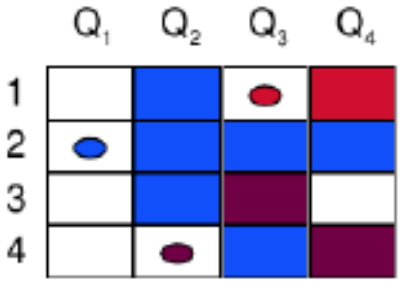
..... on remet Q1 en cause

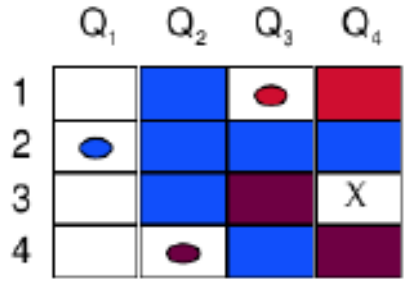
Un exemple ...suite

Après le placement de Q2, un examen entre Q3 et Q4 (anticipation)









Divers

- La génération de colonnes est souvent réalisée par les contraintes réponses.
  - ➢ Plus facile à mettre en oeuvre en CLP.
  - ◦ Une Contrainte-Réponse peut être une base pour une génération de colonnes.
  - ◦ Sur les variables restantes, on élimine certaines et on regarde la réponse.
  - ◦ cf. exemple PERT.

Quelques contraintes globales en CP

On s'intéresse en particulier à celle disponibles dans Minizinc.

- all_different(séquence) : tous les éléments 2 à 2 disjoints
- all_equal(séquence) : tous les éléments identiques
- all_disjoint(séquence_of_sets) : tous les ensembles 2 à 2 disjoints
- element(N, Séquence, Val)
- $cumulative((s_1, .., s_n), (d_1, ..., d_n), (r_1, ..., r_n), L)$
- at_list(N, Séquence, Val)
- at_most(N, Séquence, Val)
- exactly(N, Séquence, Val)
- global_cardinality(Séquence1, Séquence2, Séquence3) : tout élément $Sequence2[i]$ est présent $Sequence3[i]$ fois dans *Séquence1*
- etc...

Contrainte globale element dans CP

• $element(I, Liste, X)$ contraint la variable X à être égale à la I^{eme} variable (depuis 1) de Liste.

➙ $I \in 1..n$ peut donc être un indice

Exemples :

• Pour qu'une variable X reçoive (dans la solution à venir) une valeur dans un certain ensemble $E = \{V_1, \cdots, V_n\}$ de valeurs inconnues (pour l'instant), on peut écrire :

$I :: 1..n$, la valeur n= taille de l'ensemble est connue.

$element(I, [V_1, \cdots, V_n], X)$

• Une contrainte comme $C_J \leq 5$ peut être implantée par :

$Z \leq 5, J \in 1..n$

$element(J, (C_1, \cdots, C_n), Z)$

➙ affectera (équationnel) à Z la J^{eme} valeur dans la liste $(C_1, \cdots, C_n)$.

Exemple contrainte element

- 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=*Liste des sites ordonnée*
  - Domaine : $S_i \in 1..7$ (les 7 jours).
  - Contraintes :
    - Visiter chaque site une seule fois : alldifferent(Z)
    - Espacement des visites : pour toute paire de sites S_i et S_j consécutifs : $S_i = S_j + X$, $I \in 1..4$, $element(I, [-1, -2, 1, 2], X)$.

- Et si Z non ordonnée ?

Exemple contrainte element ...suite

N.B. : on pourrait utiliser des valeurs nominales (lundi, mardi, ...).
➟ on peut aussi exprimer la même chose avec des disjonctions (coûteuses).

• **Les propagations** :

➟ Pour 5 sites, on obtient après propagation faible :
$S_1 \in 5..7,\ S_2 \in 4..6,\ S_3 \in 3..5,\ S_4 \in 2..4,\ S_5 \in 1..3$

➟ Pour 6 sites, on obtient après propagation faible :
$S_1 \in 6..7,\ S_2 \in 5..6,\ S_3 \in 4..5,\ S_4 \in 3..4,\ S_5 \in 2..3,\ S_6 \in 1..2$

➟ Pour 7 sites, les propagation (sans énumération) place un site par jour de la semaine : $Z = 1, 2, ..., 7$.

• Il suffit ensuite de générer des valeurs pour Z.
➟ Dès qu'un site prend une valeur, on élimine les valeurs incompatibles des autres.

Contrainte Cumulative

La contrainte Cumulative :

- Utilisée pour la planification de toutes sortes.
  En particulier avec des contraintes de ressources.

- Exemple :
  - Soit n tâches $t_1, .., t_n$
  - $s_1, .., s_n$ contient les dates de débuts (*start*) : s_i est date de début de t_i
  - Les n tâches à planifier avec des durées respectives $d_1, ..., d_n$
  - Ces tâches réclament des ressources respectives : $r_1, ..., r_n$

  ➙ Le total des ressources consommées par ces tâches ne doit pas dépasser (à aucun moment) la limite L.

- On écrira :

  $$cumulative((s_1, .., s_n), (d_1, ..., d_n), (r_1, ..., r_n), L)$$

../..

Contrainte Cumulative ...suite

- Notion de couverture (*overlap*).
- Soit deux tâches $t_i : s_i..f_i (f_i \geq s_i + d_i)$ et $t_j : s_j..f_j$
  ➙ 9 cas de figures possibles. On peut savoir sans quel cas de figure se trouve-t-on :

  Si $s_i \geq s_j$
  Si $f_i \leq f_j$
  Alors $overlap = f_i - d_j$
  Sinon $overlap = f_j - d_j$
  Sinon
  Si $f_i \leq f_j$
  Alors $overlap = f_i - d_i$
  Sinon $overlap = f_j - d_i$

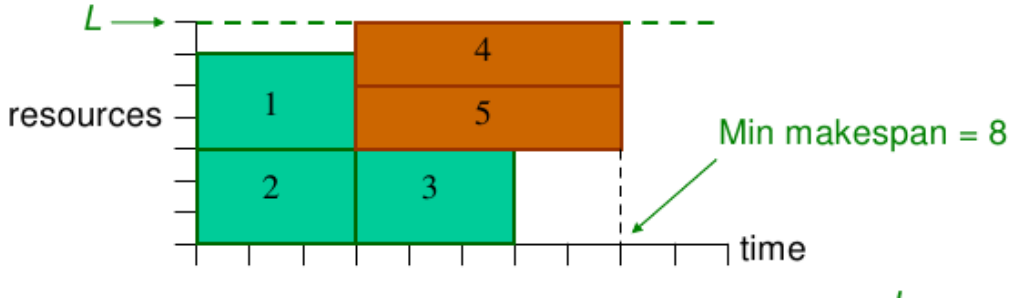
- Approche contraindre-générer : pour chaque unité du temps $u : u_s..uf$:
  ➙ Vérifier que les ressources pendant l'$overlap < Limite$

- Approche générer-tester :
  ➙ si u_s ne correspond pas à un s_i (ou u_f à un f_i) alors le test est inutile.
  ➙ L'unité du temps peut donc être une partie (> 1 *unit*) d'une tache.

Cumulative : Exemple 1

- Minimiser le temps total (*makespan*) de réalisation des taches sans deadline.

$$
\begin{array}{ll}
min & Z \\
s.t. & cumulative((s_1, .., s_5), \\
 & \quad (3,3,3,5,5), (3,3,3,2,2), 7) \\
 & Z \geq t_1 + 3 \\
 & \ldots\ldots \\
 & Z \geq t_5 + 2
\end{array}
$$



Avec :

$(s_1, .., s_5)$: 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)

☞ Une solution en MIP (ne dispose pas de *cumulative*) est donnée en Addendum.

Cumulative : Exemple 2

Exemple 2 (du manuel OPL):

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 certains nombre d'ouvriers (cf. tableau).

• On dispose de 8 ouvriers.

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

Cumulative : Exemple 2 ...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	

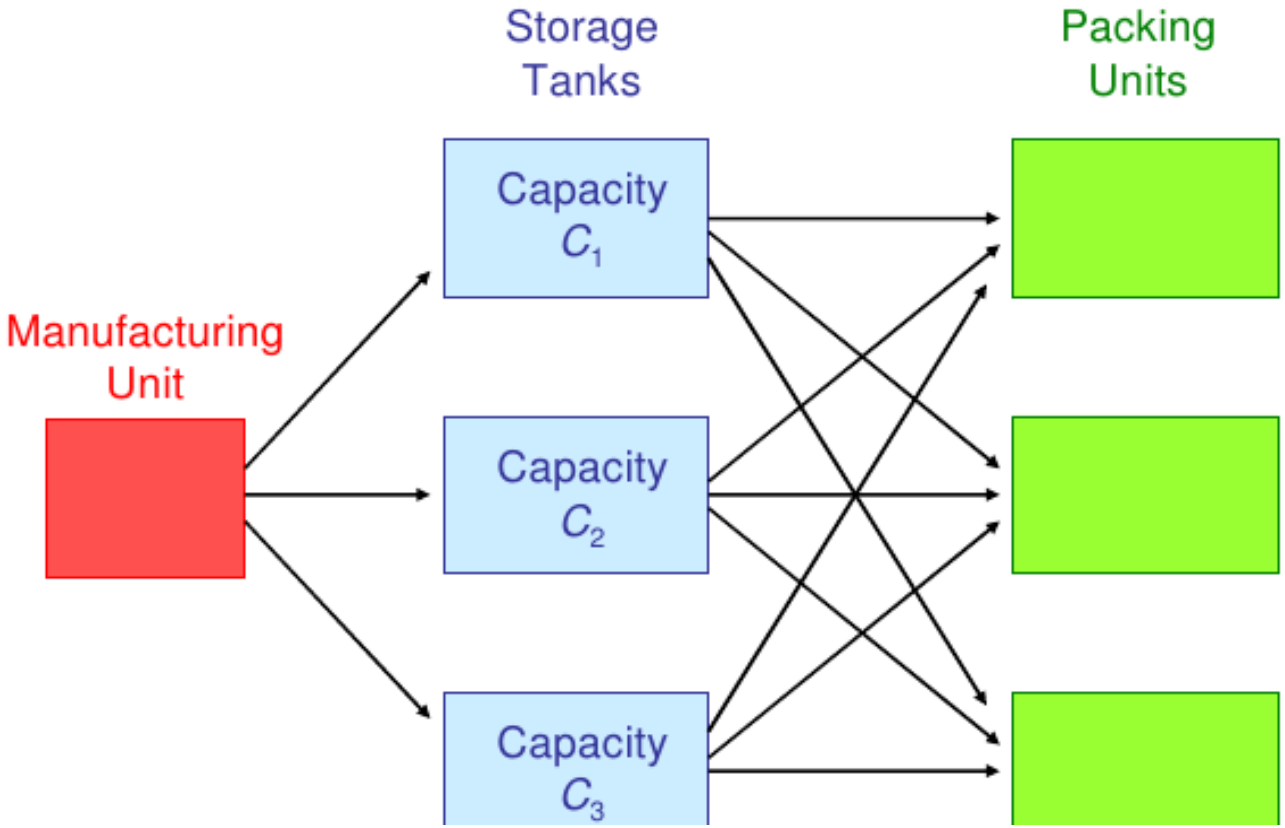
$t_i \rightarrow t_j$: t_j commence après la fin de t_i.

- Le modèle CP :

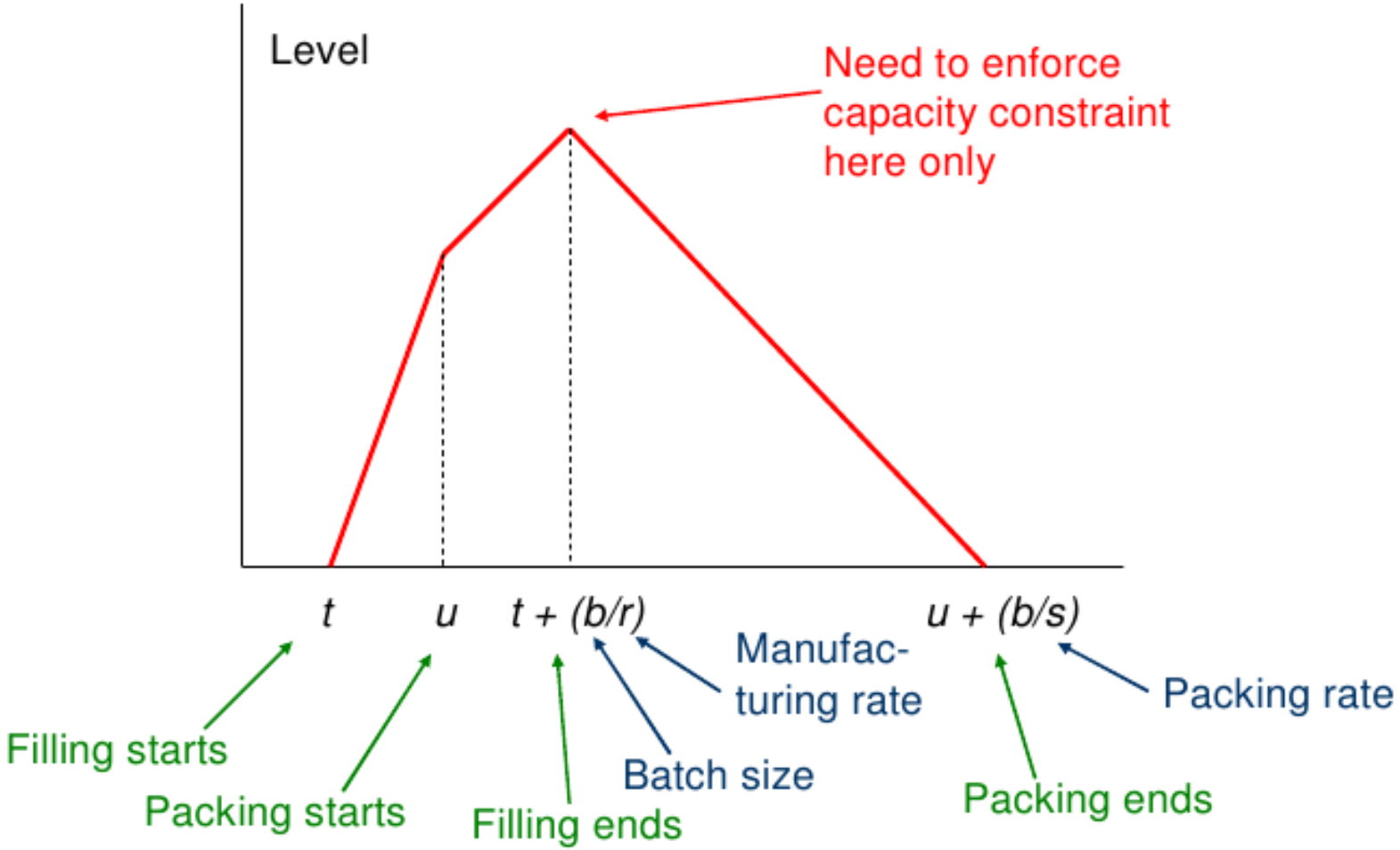
$$
\begin{array}{ll}
min & Z \\
s.t. & cumulative((t_1, .., t_{34}), \\
 & (3, 4, 2, \cdots, 2), (4, 4, \cdots, 3), 8) \\
 & Z \geq t_1 + 3 \\
 & Z \geq t_2 + 4 \\
 & \cdots\cdots \\
 & t_2 \geq t_1 + 3 \\
 & t_4 \geq t_1 + 3 \\
 & \cdots\cdots
\end{array}
$$

Cumulative : Exemple 3

Exemple 3 planification de production avec stockage intermédiaire :



Cumulative : Exemple 3 ...suite



Cumulative : Exemple 3 ...suite

$$
\begin{aligned}
min \quad & T && \leftarrow (makespan)\, A\ minimiser \\
s.t. \quad & T \geq u_j + \frac{b_j}{s_j},\ \forall j \\
& t_j \geq R_j,\ \forall j && \leftarrow temps\ fin\ tache\ t_j \\
& cumulative(t, v, e, m) && \leftarrow m\ tankers\ de\ stockage \\
& v_i = u_i + \frac{b_i}{s_i} - t_i,\ \forall i && \leftarrow duree\ des\ taches\ t_i \\
& b_i \left(1 - \frac{s_i}{r_i}\right) + s_i u_i \leq C_i,\ \forall i && \leftarrow capacite\ du\ tanker\ i \\
& cumulative\left(u, \left(\frac{b_1}{s_1}, \cdots, \frac{b_n}{s_n}\right), e, p\right) && \leftarrow p\ units\ de\ paquetage \\
& u_j \geq t_j \geq 0 && e = (1, \cdots, 1)
\end{aligned}
$$

alldifferent et circuit : Modélisation TSP

Voyageur de Commerce

Soit C_{ij} = distance de la ville i à la ville j.

➙ Trouver le chemin le + court qui visite toutes les n villes exactement 1 seule fois.

Un modèle binaire (OR):

- Soit $X_{ij} = 1$ si la ville i précède immédiatement la ville j, 0 sinon.

$$
\begin{aligned}
min \quad & \sum_{ij} C_{ij}\ X_{ij} \\
s.t. \quad & \sum_{i} X_{ij} = 1, \forall j \\
& \sum_{j} X_{ij} = 1, \forall i \\
& \sum_{i \in V} \sum_{j \in W} X_{ij} \geq 1, \forall (V, W)\, disjoint \subset \{1..n\} \\
& X_{ij} \in \{0, 1\}
\end{aligned}
$$

Les contraintes $\sum_{i \in V} \sum_{j \in W} X_{ij} \geq 1$ permet d'éliminer les circuits.

alldifferent et circuit : Modélisation TSP ...suite

Modèle CP (avec *alldifferent*) :

• Soit y_k la kième ville visitée.

$$\begin{aligned} min \quad & \sum_k C_{y_k y_{k+1}} \\ s.t. \quad & alldifferent(y_1, ..., y_n) \\ & y_k \in \{1..n\} \end{aligned}$$

- Dans $C_{y_k y_{k+1}}$, l'expression $y_k y_{k+1}$ est un indice double de variables (de la forme C_{ij})
- La contrainte $alldifferent(y_1, ..., y_n)$ est une contrainte **globale**.

alldifferent et circuit : Modélisation TSP ...suite

Un autre modèle CP (avec *circuit*)

• Soit y_k la ville visitée après la ville k.

$$\begin{aligned} min \quad & \sum_k C_{ky_k} \\ s.t. \quad & circuit(y_1, ..., y_n) \\ & y_k \in \{1..n\} \end{aligned}$$

- Dans C_{ky_k}, l'expression ky_k est un indice double de variables (de la forme C_{ij})

- La contrainte $circuit(y_1, ..., y_n)$ est une contrainte **globale** qui permet de réaliser un circuit *Hamiltonien* (un tour).

➟ Elle même est réalisée (boite à outils) à base d'autres contraintes CP.

Modélisation Infirmières et contraintes globales

- 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 en 2 shifts différents sur 2 jours consécutifs.
- Une infirmière qui travaille en shift 2 ou 3 doit faire de même au moins deux jours d'affiler.

	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 infirmières aux shifts.

	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

Exemple d'assignation des shifts aux infirmières.

Modélisation Infirmières et contraintes globales ...suite

- On utilisera les deux formulations dans le même modèle.

➙ Chaque ligne = 1 shift, chaque colonne= 1 jour de la semaine

- On utilise la matrice W : $Shifts \times Days$
- Pour chaque jour, des infirmières différentes
  $\forall colonne\ D(=1\ jour), all\_different(D)$
- Toute infirmière travaille au moins 5 jour (et maxi 6)
  $\forall ligne\ S, card(A \in S) = 5..6$, idem pour les nurses $B..D$
- Chaque shift contient 2 infirmières
  $\forall ligne\ S, nvalue(2, S)$: 2 valeurs distincts dans S
- Une infirmière ne travaille pas en 2 shifts différents sur 2 jours consécutifs.
  Pour toute case $W_{s,d}$, $W_{s,d} \neq W_{s+1,d+1} \neq W_{s+2,d+1} \neq W_{s+1,d-1} \neq W_{s+2,d-2}$
- Dernière contrainte avec Stretch (pas dans MiniZinc)

$\text{alldiff}(w_{1d}, w_{2d}, w_{3d})$, all d

$\text{cardinality}(w \mid (A,B,C,D),(5,5,5,5),(6,6,6,6))$

$\text{nvalues}(w_{s,\text{Sun}},\ldots,w_{s,\text{Sat}} \mid 1,2)$, all s

$\text{alldiff}(y_{1d}, y_{2d}, y_{3d})$, all d

$\text{stretch}(y_{i,\text{Sun}},\ldots,y_{i,\text{Sat}} \mid (2,3),(2,2),(6,6),P)$, all i

$w_{y_{id}d} = i$, all i,d

$y_{w_{sd}d} = s$, all s,d

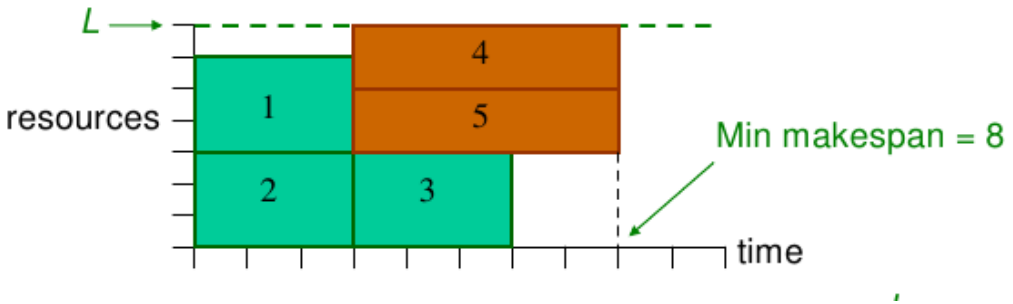
- *Stretch(Vecteur, Valeurs, P)* : impose qu'une séquence consécutive maximale des variables dans *Vecteur* prennent des valeurs dans *Valeurs* (résultat dans P).
- *nvalues(Nb, Vecteur)* : impose que Nb valeurs distinctes se trouvent dans *Vecteur*.
- *cardinality(Vecteur, Valeur, Nb)* : contrainte sur le Nb d'occurrences d'une Valeur dans un Vecteur.

- Voir solution MiniZinc.

Solution Exemple 1 cumulative

- Minimiser le temps total (*makespan*) de réalisation des taches sans deadline.

$$
\begin{array}{ll}
min & Z \\
s.t. & cumulative((t_1, .., t_5), \\
 & (3,3,3,5,5), (3,3,3,2,2 \\
 & Z \geq t_1 + 3 \\
 & \ldots\ldots \\
 & Z \geq t_5 + 2
\end{array}
$$



Avec :

$(t_1, .., t_5)$: 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)

Solution Exemple 1 cumulative ...suite

Solutions MIP :

- On peut considérer le temps discret et considérer des unités de temps = 1.
- Il existe une solution complexe si on veut considérer un temps continue.

- Pour la fenêtre du temps, on peut penser prendre le min des durées au lieu de 1,
  - Non, car toute tâche peut commencer à n'importe quel moment et on sera donc obligé de considérer chaque unité, même si dans la solution proposée (c'est une des solutions), on peut considérer la fenêtre temporelle = 3.
  - Compromis : considérer une fenêtre=min des durées, c'est imposer une contrainte supplémentaire et éviter certaines solutions.
    ➛ On peut tester pour voir si il y a une solution.

Solution Exemple 1 cumulative ...suite

Une solution (MLP) sous LPSOLVE :

Raisonnement :

• On considère des unités du temps (discret) et on se donne 10 unités (1..10). On peut prendre 0..10 pour commencer à 0 (mais je suppose qu'on commence à l'unité 1).

◦ Soit $b_{1,1} = 1$: si t_1 active pendant u_1, 0 sinon.

◦ De même, $b_{1,2} = 1$: si t_2 active pendant u_1, 0 sinon.

◦ ... $b_{1,10}$ ...

• On peut alors noter pour l'unité du temps 1 (u_1) :

$$3b_{1,1} + 3b_{2,1} + 3b_{3,1} + 2b_{4,1} + 2b_{5,1} <= 7;$$

➙ C-à-d : la somme des ressources utilisées par les 5 tâches pendant u_1 $<= 7$

• On fait de même pour l'unité u_2, ...u_{10}

• Ensuite, on doit dire : 3 unités parmi 10 suffisent pour t_1, ...

➙ $b_{1,1} + b_{1,2} + b_{1,3} + b_{1,4} + b_{1,5} + b_{1,6} + b_{1,7} + b_{1,8} + b_{1,9} + b_{1,10} = 3$

• De même pour les autres tâches.

Solution Exemple 1 cumulative ...suite

```
min: ;
    b11+b12+b13+b14+b15+b16+b17+b18+b19+b110 =3;
    b21+b22+b23+b24+b25+b26+b27+b28+b29+b210 =3;
    b31+b32+b33+b34+b35+b36+b37+b38+b39+b310 =3;
    b41+b42+b43+b44+b45+b46+b47+b48+b49+b410 =5;
    b51+b52+b53+b54+b55+b56+b57+b58+b59+b510 =5;

    3 b11 + 3 b21 + 3 b31 + 2 b41 + 2 b51 <= 7;
    3 b12 + 3 b22 + 3 b32 + 2 b42 + 2 b52 <= 7;
    3 b13 + 3 b23 + 3 b33 + 2 b43 + 2 b53 <= 7;
    3 b14 + 3 b24 + 3 b34 + 2 b44 + 2 b54 <= 7;
    3 b15 + 3 b25 + 3 b35 + 2 b45 + 2 b55 <= 7;
    3 b16 + 3 b26 + 3 b36 + 2 b46 + 2 b56 <= 7;
    3 b17 + 3 b27 + 3 b37 + 2 b47 + 2 b57 <= 7;
    3 b18 + 3 b28 + 3 b38 + 2 b48 + 2 b58 <= 7;
    3 b19 + 3 b29 + 3 b39 + 2 b49 + 2 b59 <= 7;
    3 b10 + 3 b210 + 3 b310 + 2 b410 + 2 b510 <= 7;
    b110=0;
 bin  b11,b12,b13,b14,b15,b16,b17,b18,b19,b21,b22,b23,b24,b25,b26,b27,b28,b29,b210,
b31,b32,b33,b34,b35,b36,b37,b38,b39,b310,b41,b42,b43,b44,b45,b46,b47,b48,b49,b410,
b51,b52,b53,b54,b55,b56,b57,b58,b59,b510;
```

Solution Exemple 1 cumulative ...suite

- On obtient :
  - ➙ t1 : unités 6,7,10 (pas continues)
  - ➙ t2 : 4..6
  - ➙ t3 : 1..3
  - ➙ t4 : 1..5
  - ➙ t5 : 1..5
- J'ai du forcer que l'unité 10 ne soit pas utilisée pour t1 (b1,10=0) et là, j'ai une solution :
  - ➙ t1 : 6,7,8
  - ➙ t2 : 4..6
  - ➙ t3 : 1..3
  - ➙ t4 : 1..5
  - ➙ t5 : 1..5
  - ➙ C'est bon. Faire le graphe en commençant par t4 et t5, puis t3, t2 et t1.
- Ce modèle montre la démarche à suivre pour s'adapter à l'outil (ici *LpSolve*).
  - ➛ On perd en clarté et en aspect déclaratif.
  - ➛ Par contre, on est gagnant en efficacité / rapidité

☞ Ne veut pas dire que les problèmes complexes auront aussi facilement une solution de bas niveau !

CP : comparaison avec LP

• CP vs. Programmation Linéaire

LP	→	**CP / CLP**
Calcul numérique	→	Traitement de la Logique (fonctionnel / impératif présents)
Relaxation	→	Inférence (filtrage, propagation de contraintes)
Modélisation basique (inégalité linéaire)	→	Modélisation Haut-niveau (contraintes globales)
Branching	→	Branching
Independence du modèle & algo	→	Idem (traitement à base de contraintes)
(**Le comment, pas le quoi !**)	→	Idem

• En CP / CLP : les équations (contraintes) décrivent le problème, mais ne disent pas **comment** le résoudre.

◦ En LP, parfois, on est obligé de descendre à un niveau plus élémentaire (cf. l'exemple précédent et *LpSolve*)

• En CP : chaque contrainte invoque une "procédure" qui élimine des solutions inacceptables.

➙ Comme chaque ligne d'un code (impératif) invoque une opération.

CP : comparaison avec LP ...suite

Avantages de CP

- Meilleur en ordonnancement et en planification
  → ... où les méthodes MP ont des relaxations faibles
- L'ajout de contraintes redondantes rend le problème plus facile.
  → Plus il y a des contraintes, mieux c'est.
- Langage de modélisation plus puissant.
  → Les contraintes globales conduisent à des modèles succincts.
  → Les contraintes transmettent la structure du problème au solveur
- Meilleure pour les problèmes sur-contraints
  → trompeuse : mieux lorsque les contraintes se propagent bien,
  ou lorsque les contraintes n'ont pas trop de variables.
- CP : domaines $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{B}, \mathbb{S}, \mathbb{L}, ...$
  → Permettent une modélisation plus simple.
- CP englobe de plus en plus les techniques RO (via des sous-systèmes)
  → Ex : EPLEX sous Eclipse, MiniZinc & Lpsolve sous BProlog, ...

CP : comparaison avec LP ...suite

Inconvénients de CP

- Plus faible pour les variables continues.
  → En raison du manque de techniques numériques (s'améliore !)
- Peut échouer lorsque les contraintes contiennent trop de variables.
  → Et que ces contraintes ne se propagent pas bien.
- Souvent pas bon pour trouver des solutions optimales.
  → En raison de l'absence de techniques de relaxation (peut se faire !).
- Mise à l'échelle difficile
  → plutôt dans les méthodes combinatoires discrètes
  (néanmoins, travaux et progrès sur ce domaine)

• Les logiciels de plus en plus robuste.
• On va vers une intégration MP/CP/heuristiques.

• L'exemple suivant (dans $\mathbb{R}$) montre la capacité de CP / CLP à écrire des prédicats.

Exemple amortissement

Exemple d'amortissement (en *Picat / BProlog*)

```
% P: principal (amount borrowed)
% T: time periods
% I: interest rate
% R: repayment
% B: balance (final amount owing)

import mip.

mortgage(P,T,I,R,B) ?=>
  T = 0,
  B #= P,
  solve([P,T,I,R,B]).

mortgage(P,T,I,R,B) =>
  T >= 0,
  NT #= T -1,
  NP #= P + P*I -R,
  mortgage(NP,NT,I,R,B).
```

Exemple amortissement ...suite

- Quelques tests (et réponses) :

```
go =>
   mortgage(1000.0,10,10.0/100.0,150.0,B),
   println(b=B),
   nl.
% b=203.128769950000162

go2 =>
   mortgage(P,10,10.0/100.0,150.0,0),
   println(p=P),
   nl.
% p=921.685065855701964

go3 =>
   mortgage(P,10,10.0/100.0,R,B),
   println([p=P,r=R,b=B]),
   nl.
 % 0.3855*B + 6.1446*R

go4 =>
   B #= 10.0,
   mortgage(1234,10,10.0/100.0,R,B),
   println([r=R,b=B]),
   nl.
```

Exemple amortissement ...suite

- D'autres exemples

```
mortgae(Cap_init, Duree, Taux, Mensualite, Rest)

1- J'emprunte 100 et je ne veux rien rembourser !
mortgage(100, 12,0.1, 0,Reste).
Reste = 313.842837672099961

2- ici, question simple : 100 au départ, quel mensualité ?
 100 emprunté, combien par mois avec 0 à la fin.

mortgage(100,12,0.1, M,0)
      0:   objval =   0.000000000e+00   infeas =   1.000000000e+00 (11)
     12:   objval =   0.000000000e+00   infeas =   0.000000000e+00 (0)
OPTIMAL SOLUTION FOUND
M = 14.67633151002876 ?

3- pour 100 emprunté, combien je devrais à la fin si je rembourse 10 par mois ?
mortgage(100, 12,0.1, 10,Reste).

Reste = 100.0  ==> je dois encore 100  car le taux d'intérêt est par mois !

4- Combien emprunter si je paie 10 par mois
 mortgage(P,12,0.1, 10,0)
      0:   objval =   0.000000000e+00   infeas =   1.000000000e+00 (0)
     12:   objval =   0.000000000e+00   infeas =   0.000000000e+00 (0)
OPTIMAL SOLUTION FOUND
P = 68.136918228964305 ?

5- j'emprunte 68 et je paie 10 par mois, à la fin, on me doit 0.4

mortgage(68, 12,0.1, 10,Reste).
Reste = -0.429708055071984 ?
```

Cadre d'unification

• L'exemple précédent montre l'utilisation du solveur MIP sous CLP.

• De plus en plus d'environnements CP / CLP ont réalisés une interface vers les langages LP.
  ◦ Par exemple : *Prolog / Picat* vers *Minizinc / Flat Zinc* ou *GLpk*
  ◦ *Minizinc* permet également une interface vers *LpSolve*

• Ces interfaces permettent de choisir au mieux le solveur le plus efficace

• Les techniques de Programmation Mathématiques sont également directement applicables / utilisables en CP/CLP.
  ◦ On a vu plus haut quelques exemples de traduction.

• Ces interfaces sont aux CP/CLP ce qu'un langage d'assmbleur est aux langages impératifs.

• Un élément majeur de CP/CLP est la possibilité de récursivité (cf. exemple *mortgage*).
• ....

Modélisation Tourné

Exemple 1 : 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 universités sont situées dans les villes "St-Étienne", "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

Modélisation Tourné ...suite

Modélisation OR :

1. **Choix des variables** : une phase importante (clarté, simplicité, complexité).
   • Les étapes sont entre 2 villes.
   • On peut décider de représenter chaque étape par une variable $x_{dep,arr}$ dont la valeur finale reflète le fait de faire ou non une étape entre 2 villes.

2. **Les domaines** : booléen (0, 1).
   ➙ E.g. si $x_{1,2} = 1$ alors on fera l'étape entre ville1 et ville2; 0 sinon.

$$x_{i,j} \in \{0,1\} \quad \forall\ i,j = 1,2,3,4 \qquad \text{(C1)}$$

☞ Les domaines des variables peut être considéré comme une contrainte. Le choix des variables et leurs domaines est en rapport direct avec la complexité.

☞ Ne pas hésiter à changer de variable si l'on estime que le choix est pas bon !

Modélisation Tourné ...suite

3. **Modélisation et contraintes :**
   1. Il n'y aura pas d'étape d'une ville à elle même, donc :

$$x_{i,i} = 0 \quad \forall\ i = 1, 2, 3, 4 \qquad \text{(C2)}$$

   2. Chaque ville est visitée une seule fois.
   Par exemple, ville-2 (St-Étienne) ne doit apparaitre comme étape d'arrivée qu'une seule fois et donc $x_{i,2} \neq 0$ pour exactement un i.
   ➙ Ce qui donne, pour cette ville : $x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1$.
   • Généralisation : toutes les variables sont binaires (et on a la contrainte C2 qui ne gène pas) :

$$\sum_{i=1}^{4} x_{i,j} = 1,\ \forall j\ \textit{fixé} \in 1..4 \qquad \text{(C3)}$$

   ➙ Ex : on fixe $j = 2$, on fait varier $i \in 1..4$ et on obtient (sachant que $x_{1,1} = 0$) $\quad x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1.$
   ➙ Faire de même pour $j = 1, 3, 4$

Modélisation Tourné ...suite

3. Les contraintes précédentes assurent que chaque ville est visitée une seule fois (comme destination) mais elles ne garantissent pas une solution correcte.
   ➞ Par exemple, la (mauvaise) solution $x_{1,2} = 1, x_{1,3} = 1, x_{1,4} = 1, x_{2,1} = 1$ et 0 pour toutes les autres variables satisfera les contraintes ci-dessus et pourtant, ce parcours-là est improbable !

   • Pour régler ce problème, il nous faut éviter qu'une ville soit plus d'une fois l'origine d'une étape et ajouter la contrainte :

$$\sum_{j=1}^{4} x_{i,j} = 1, \ \forall i \ \textit{fixé} \in 1..4 \qquad \text{(C4)}$$

   ➞ Par exemple, pour i fixé à 2, on impose que $x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} = 1$, c'est à dire : une seule de ces 4 variables vaudra 1.

Modélisation Tourné ...suite

4. Les étapes obtenues ne doivent pas être déconnectées les unes des autres.
   ➙ Par exemple, une solution telle que $x_{1,2} = 1, x_{2,1} = 1, x_{3,4} = 1, x_{4,3} = 1$ (et toutes les autres=0) ne constitue pas un voyage aller-retour.
   ➙ Nous pouvons utiliser (appelée en CP une étape *vérification de la solution*)

$$x_{i,j} + x_{j,i} \leq 1, \ \forall i = 1..4, j = 1..4 \qquad \text{(C5)}$$

   ☞ Pourquoi ne pas utiliser ici $x_{i,j} + x_{j,i} = 1, \ \forall ...$?
   ➥ Car par ex. pour $i = j = 2$, on aura $x_{2,2} + x_{2,2} = 0$.
   ➥ Aussi, imposer $x_{i,j} + x_{j,i} = 1$ (au lieu de ≤ 1) exigera qu'au moins l'une des deux variables ($x_{i,j}$ ou $x_{j,i}$) soit $= 1$. Or, il se peut que dans certaines combinaisons i, j, les deux variables soient nulles dans la solution finale.

• L'analyse OR ci-dessus est complète. Il suffira d'écrire les contraintes BIP (telles que indiquées ci-dessus pour obtenir une solution.
➙ Il y aura environ 30 expressions.

Modélisation Tourné ...suite

- **Solution OR (lp_solve) :**

```
min : ;

/* Contrainte C1 */
X11=0; X22=0;X33=0;X44=0;

/* Contrainte C3 (départs) */
X12+X11+X13+X14=1;
X11+X13+X14+X12=1;
X13+X14+X12+X11=1;
X14+X12+X11+X13=1;
X22+X21+X23+X24=1;
X21+X23+X24+X22=1;
X23+X24+X22+X21=1;
X24+X22+X21+X23=1;
X31+X32+X33+X34=1;
X32+X33+X34+X31=1;
X33+X34+X31+X32=1;
X34+X31+X32+X33=1;
X41+X42+X43+X44=1;
X42+X43+X44+X41=1;
X43+X44+X41+X42=1;
X44+X41+X42+X43=1;


/* Contrainte C3 (arrivées) */
X22+X12+X32+X42=1;
X21+X11+X31+X41=1;
X13+X23+X33+X43=1;
X14+X24+X34+X44=1;
X12+X22+X32+X42=1;
X11+X21+X31+X41=1;
X23+X13+X33+X43=1;
```

Modélisation Tourné ...suite

```
 X24+X14+X34+X44=1;
 X32+X22+X12+X42=1;
 X31+X21+X11+X41=1;
 X33+X13+X23+X43=1;
 X34+X14+X24+X44=1;
 X42+X22+X12+X32=1;
 X41+X21+X11+X31=1;
 X43+X13+X23+X33=1;
 X44+X14+X24+X34=1;

 /* faire une tournée  */
 X12+X21 <= 1;
 X13+X31 <= 1;
 X14+X41 <= 1;
 X23+X32 <= 1;
 X24+X42 <= 1;
 X34+X43 <= 1;

int X12,X13,X14,X21,X22,X23,X24,X31,X32,X33,X34,X41,X42,X43,X44;
```

- Solution lp_solve :

```
X12                           1
X24                           1
X31                           1
X43                           1

Toutes les autres variable = 0
```

Modélisation Tourné ...suite

Solution CP :

- En CP, on peut coder cet exemple exactement comme avec OR.
  ➙ Les systèmes CP peuvent utiliser un solveur Simplex à la demande.
- Une solution sans passer par un MIP.

➙ **Choix des variables :**

```
Liste_etapes=[X11,X12,X13,X14,X21,X22,X23,X24,X31,X32,X33,X34,X41,X42,X43,X44],
```

➙ **Domaine des variables :**

```
Liste_etapes   ::  0..1,
```

➙ **Contrainte C1 : on va pas d'une ville à elle même**

```
X11 = 0, X22 = 0, X33 = 0, X44 = 0,

Liste_Depart_ville_1=[X11,X12,X13,X14],        Liste_Depart_ville_2=[X21,X22,X23,X24],
Liste_Depart_ville_3=[X31,X32,X33,X34],        Liste_Depart_ville_4=[X41,X42,X43,X44],
```

➙ **Contrainte C3 : une ville est seulement une fois le départ d'une étape :**

```
only_one_true (Liste_Depart_ville_1),
only_one_true(Liste_Depart_ville_2),
only_one_true(Liste_Depart_ville_3),
only_one_true(Liste_Depart_ville_4),

Liste_Arrivee_ville__1=[X11,X21,X31,X41],        Liste_Arrivee_ville__2=[X12,X22,X32,X42],
Liste_Arrivee_ville__3=[X13,X23,X33,X43],        Liste_Arrivee_ville__4=[X14,X24,X34,X44],
```

➙ **Contrainte C2 : une ville est seulement une fois l'arrivée d'une étape :**

```
only_one_true(Liste_Arrivee_ville__1),
only_one_true(Liste_Arrivee_ville__2),
only_one_true(Liste_Arrivee_ville__3),
only_one_true(Liste_Arrivee_ville__4),
```

Modélisation Tourné ...suite

➙ **Contrainte C4 : que_ce_soit_une_tourne**

```
X12   NAND  X21,  X13 NAND  X31,  X14 NAND  X41,
X23 NAND  X32,  X24 NAND  X42,
X34 NAND  X43.

enumerer (Liste_etapes).
```

- Solutions : notons qu'en CP, on obtient toutes les solutions.

➙ X14=1, X23=1, X31=1, X42=1 (V1 → V4 → V2 → V3 → V1)

➙ X14=1, X21=1, X32=1, X43=1.

➙ X13=1, X24=1, X32=1, X41=1.

➙ X13=1 ,X21=1, X34=1, X42=1.

➙ X12=1, X24=1, X31=1, X43=1.

➙ X12=1, X23=1, X34=1, X41=1.

Modélisation CP : Tournoi

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.

Modélisation CP : Tournoi ...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 différents idées de solution ../..

Modélisation CP : Tournoi ...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 ? (variables 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

Modélisation CP : Tournoi ...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

Modélisation CP : Tournoi ...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 matchs

  ➞ 56 contraintes sur chacune des7 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 !

Modélisation CP : Tournoi ...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 coté des contraintes non-domaine-fini ?

☞ cf. exemple de choix structure de donnes dans l'exemple 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 x 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

1. **Choix des variables :**
   Une matrice carré **M** de $7 \times 7 = 49$ de couples d'équipes ($Jours \times Site$)
   ➞ Chaque couple dans une cellule $Mij, i, j = 1..7$
2. **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 $[0, 0]$.
3. **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.

Une solution CP ...suite

```
Tournoi(M) :
    Créer_matrice(M[7,7]) contenant 49 couples [Ei,Ej]
    Fixer_Domaines : pour chaque [Ei,Ej] = 0..8, 0..8    ☞ 0 pour [0,0]
    Contraindre_lignes(M) : chaque case d'une ligne contient un couple [Ei,Ej] ou [0,0]
                ☞       [Ei, Ej], Ei/=Ej est unique mais pas les [0,0]
    Contraindre_colonnes(M) : chaque case d'une colonne contient un couple [Ei,Ej] ou [0,0]
                ☞       [Ei, Ej], Ei/=Ej est unique mais pas les [0,0]
    enumerer(M).

Contraindre_lignes(M) :
    Pour toutes les lignes L de M
        Toute_equipe_0_ou_1_fois_dans_chaque_ligne(L)

Contraindre_colonnes(M) :
    Transposer(M,M')        % pour faire de même que pour les lignes
    Pour toutes les lignes L de M'
        Toute_equipe_0_ou_1_fois_dans_chaque_ligne(L)

Toute_equipe_0_ou_1_fois_dans_une_ligne(L) :
          ☞  L = [E1,E2]_1, ... , [E1,E2]_7
          ☞  imposer à chaque [E1,E2]_i d'être soit =[0,0], soit unique dans L

    Tout_couple_contient_2_equipes_ou_0_0(L),
    Couples_tous_differents_sauf_pour_0_0(L),
    Toute_equipe_est_placee(L).      ☞ Ici, on est déconnecté des colonnes et on veut
```

Une solution CP ...suite

```
                                                que toutes les équipes soient présentes
Couples_tous_differents_sauf_pour_0_0(L : [E1,E2]_1, ... , [E1,E2]_7) :
    Aplatir L ➙  L'= [E11,E21, E12,E22, E13,E23, ... E77] : 14 variables
    Pour tout X dans L'
        X_egal_0_ou_pas_dans(X, reste_de(L')),
    Couples_tous_differents_sauf_pour_0_0(reste_de(L')). ☞ à ne  pas oublier


X_egal_0_ou_pas_dans(X, L) :      % L est une liste plate
    Pour tout élément Y de L      % Si L=[], ne rien faire
        (X = 0 <==> true)
         OR
        (X /=0 <==> X /= Y).

Tout_couple_contient_2_equipes_ou_0_0(L : L = [X,Y]_1, ... , [X,Y]_7) :
    Pour tout couple <X,Y> dans L :
       (X  = 0  <==> Y = 0)
       OR
       (X /= 0  <==> Y /= 0).       % pas besoin de dire X /=Y (on pourrait) car :
    ☞  De plus, la somme = 36 impose que tous les chiffres (équipes) soient présents.

Toute_equipe_est_placee(L) :     %  est appelé "vérification d'une solution en CP"
    % par exemple :
    Somme_tous_chiffres(L,36)  %(car 1+2+...+8=36)
```

Une solution CP ...suite

Remarques :

• La contrainte exactement_un existe pour les variables en domaine fini
  ➙ Mais on peut créer la notre (par exemple pour un match).
  ➙ cf. *alldifferent* sur les couples d'équipes (utilisé ci-dessus)

• Dans certains environnements, il est possible de transformer un prédicat en contrainte (cf. Eclipse).
  ➙ Expliquer la différence.
  ➙ Eclipse propose également *gcc : global cardinality constraint*
  ➙ *gcc(low, high, value)* : *value* $\in$ *low..high*

Modélisation Ateliers

ZZ : placer dans les exercices BE (pour 17-18)
➟ J'ai la solution (à vérifier) dans Mes-Xes-MiniZinc/Ex4-ateliers3.mzn
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, ...

- *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*

Contraintes Globales de Minizinc

(voir par famille plus loin)

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 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 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)
  → Constraints 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 ∈ 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 a j ∈ f[i] if and only if i ∈ 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 <1... Q, 1... S> 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], ..., 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 ∈ 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) $\Longleftrightarrow$ (i 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 in s $\Leftrightarrow$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 occurences 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 occurences 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 occurences 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 occurences 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 dimensios
  - 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 dimensios
  - 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 dimensios
  - 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) -> 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) -> 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.

Addendum : Construction empirique du Dual

- Un exemple de construction du Dual par l'exemple.
  ➙ Si vous n'aimez pas la version formelle !
- Soit le système primal :

$$
\begin{aligned}
max \quad & Z = 4x1 + x2 + 5x3 + 3x4 \\
s.t. \quad & x1 - x2 - x3 + 3x4 \leq 1 \\
& 5x1 + x2 + 3x3 + 8x4 \leq 55 \\
& -x1 + 2x2 + 3x3 - 5x4 \leq 3 \\
& x1, x2, x3, x4 \geq 0
\end{aligned}
$$

1) On produit le système

$$
\begin{aligned}
max \quad & Z = 4x1 + x2 + 5x3 + 3x4 \\
s.t. \quad & y1*(x1 - x2 - x3 + 3x4 \leq 1) \\
& y2*(5x1 + x2 + 3x3 + 8x4 \leq 55) \\
& y3*(-x1 + 2x2 + 3x3 - 5x4 \leq 3) \\
& x1, x2, x3, x4 \geq 0
\end{aligned}
$$

Addendum : Construction empirique du Dual ...suite

Addendum : Construction empirique du Dual ...suite

Rappel (système primal):

$$
\begin{aligned}
max \quad & Z = 4x1 + x2 + 5x3 + 3x4 \\
s.t. \quad & +1x1 - x2 - x3 + 3x4 \leq 1 \\
& +5x1 + x2 + 3x3 + 8x4 \leq 55 \\
& -1x1 + 2x2 + 3x3 - 5x4 \leq 3 \\
& x1, x2, x3, x4 \geq 0
\end{aligned}
$$

2) factorisation sur les variables primale (on additionne les membres):

$$
\begin{aligned}
& (+1y1+5y2-1y3) * x1 \\
& +(-y1 + y2 + 2y3) * x2 \\
& +(-y1 + 3y2 + 3y3) * x3 \\
& +(3y1 + 8y2 - 5y3) * x4 \leq (1y1 + 55y2 + 3y3)
\end{aligned}
$$

Addendum : Construction empirique du Dual ...suite

Rappel (de 2) :

$(+1y1+5y2-1y3) * x1 + (-y1 + y2 + 2y3) * x2 + (-y1 + 3y2 + 3y3) * x3 + (3y1 + 8y2 - 5y3) * x4 \leq (1y1 + 55y2 + 3y3)$

Rappel de l'objective : $max \quad Z = 4x1 + x2 + 5x3 + 3x4$

3) La condition pour que le membre gauche soit $> Z = +4x1 + 1x2+5x3+3x4$

$(+1y1+5y2-1y3) \geq 4$

$-y1 + y2 + 2y3 \geq 1$

$-y1 + 3y2 + 3y3 \geq 5$

$3y1 + 8y2 - 5y3 \geq 3$

4) Rappel (du système primal) : $maximiser \quad Z = 4x1 + x2 + 5x3 + 3x4$

$4x1 + x2 + 5x3 + 3x4 \leq z \leq 1y1 + 55y2 + 3y3$

Addendum : Construction empirique du Dual ...suite

- On obtient le système dual final :

$$
\begin{array}{rr}
\textit{Minimser} & Z' = y1 + 55y2 + 3y3 \\
s.t. & y1 + 5y2 - 1y3 \geq 4 \\
& -y1 + y2 + 2y3 \geq 1 \\
& -y1 + 3y2 + 3y3 \geq 5 \\
& 3y1 + 8y2 - 5y3 \geq 3 \\
& y_i \geq 0
\end{array}
$$

Rappel du problème primal :

$$
\begin{array}{rr}
\textit{Maximiser} & Z = 4x1 + x2 + 5x3 + 3x4 \\
s.t. & x1 - x2 - x3 + 3x4 \leq 1 \\
& 5x1 + x2 + 3x3 + 8x4 \leq 55 \\
& -x1 + 2x2 + 3x3 - 5x4 \leq 3 \\
& x1, x2, x3, x4 \geq 0
\end{array}
$$

Addendum : Construction empirique du Dual ...suite

Addendum : Un autre Dual par construction

- Soit le système primal :

$$\begin{aligned} max \quad & Z = 2x1 + x2 \\ s.t. \quad & x1 - x2 < 1 \\ & x1 < 2 \\ & x2 < 2 \end{aligned}$$

1) on harmonise les contraintes pour mieux comprendre les opérations :

$$\begin{aligned} max \quad & Z = 2x1 + x2 \\ s.t. \quad & y1(x1 - x2 < 1) \\ & y2(x1 + 0\ x2 < 2) \\ & y3(0\ x1 + x2 < 2) \end{aligned}$$

2) Factorisations sur x_i (addition des membres)

$$(1\ y1 + 1\ y2 + 0\ y3)x1 + (-1\ y1 + 0\ y2 + y3)x2 < (1\ y1 + 2\ y2 + 2\ y3)$$

Addendum : Un autre Dual par construction ...suite

3) Remarquer que les variables deviennent des contraintes
$(1\ y1 + 1\ y2 + 0\ y3) > 2$
$(-1\ y1 + 0\ y2 + y3) > 1$

4) $2x1 + x2 \leq z \leq 1\ y1 + 2\ y2 + 2\ y3$

- Le système Dual (extremum = 6):

$$\begin{array}{ll} min & Z' = y1 + 2y2 + 2y3 \\ s.t. & y1 + y2 > 2 \\ & -\ y1 + y3 > 1 \end{array}$$

Rappel du Primal (extremum = 6) :

$$\begin{array}{ll} max & Z = 2x1 + x2 \\ s.t. & x1 - x2 < 1 \\ & x1 < 2 \end{array}$$

Addendum : Un autre Dual par construction ...suite

$$x2 < 2$$

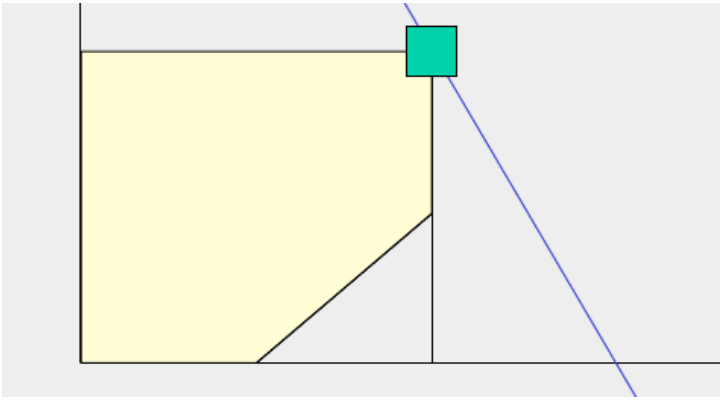
N.B. : si on cherche à simplifier le Primal, par exemple en regroupant les contraintes $x1 < 2$ et $x2 < 2$, on obtiendrait un système équivalent.

➟ Par contre, sans cet regroupement, il faut conserver la manière par laquelle les variables duales yi ont été introduites ci-dessus.

• Graphique du Primal ($x1 = x2 = 2$, extremum $= 6$)

Rappel du Primal (extremum $= 6$) :

$$\begin{array}{ll} max & Z = 2x1 + x2 \\ s.t. & x1 - x2 < 1 \\ & x1 < 2 \\ & x2 < 2 \end{array}$$



Addendum : un autre exemple Primal-Dual

- Primal :

$$\begin{aligned} Min \quad & Z = 4x1 + 7x2 \\ s.t. \quad & 2x1 + 3x2 \geq 6 \\ & 2x1 + x2 \geq 4 \\ & x1, x2 \geq 0 \end{aligned}$$

- Le Dual obtenu :

$$\begin{aligned} Max \quad & Z' = 6u1 + 4u2 \\ s.t. \quad & 2u1 + 2u2 \leq 4 \\ & 3u1 + u2 \leq 7 \\ & u1, u2 \geq 0 \end{aligned}$$

- Primal :$(x1, x2) = (3, 0)$ et extremum =12.
- La solution du Dual : $(u1, u2) = (2, 0)$
  ➙ et $Z' = 12$ **toujours le même extremum**.

Addendum : un autre exemple Primal-Dual ...suite

• Ici, le Primal cherche **la plus petite valeur** pour Z alors que le Dual cherche **la plus grande valeur** pour Z' telle que $Z \leq Z'$

• Pour comprendre cette dualité, voir l'exemple du commerçant et les matières premières (ci-après).

➟ Une autre manière de comprendre est de se placer de part et d'autre de la ligne hachurée dans la figure (graphique du Primal) :

• Observation : $4x1 + 6x2 \geq 12$ peut être obtenu depuis le Primal par $2x1 + 3x2 \geq 6$ (en la multipliant par 2).

➟ On constate que $4x1 + 6x2 \geq 12$ **domine** $Z = 4x1 + 7x2$ en notant que $4x1 + 6x2$ est la même que l'expression Z' du Dual.

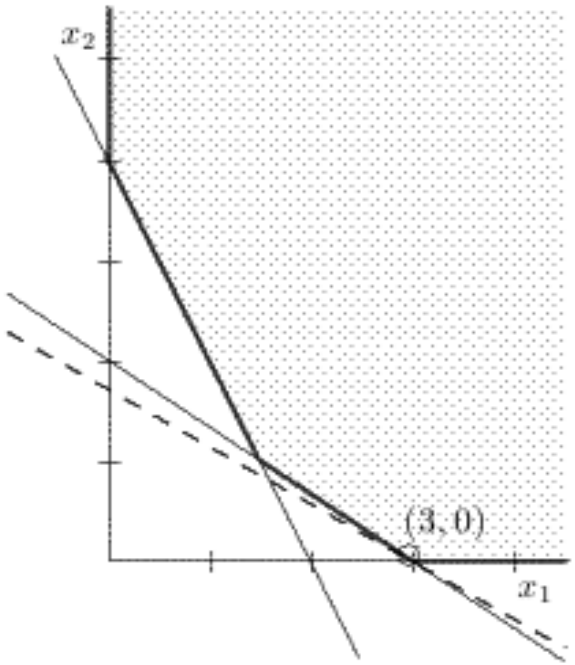
➟ Le rôle du Dual est de chercher la plus grande inégalité $cx \geq v$ qui domine l'extremum du Primal .

• Fig : la partie hachurée détermine la (zone *faisable*).

• Le petit cercle donne la solution (3, 0) du Primal;

• La ligne en pointillé représente la plus forte inégalité de la forme $4x1+7x3 \geq \alpha$ qui peut être déduite de l'ensemble des contraintes (i.e., $\alpha = 12$).

• La solution Duale est une *preuve* qui produit $4x1 + 7x3 \geq 12$



Addendum : Primal complet du Pb. de Bernard

Solution Minizinc

- On modélise et obtient d'abord une solution de la forme primale :

```
% volume litres de chaque Cocktail
array [1..4] of var float : Tab_litre_Cocktail;
var float : Ltr_Poire_used;
var float : Ltr_Pomme_used;
var float : Ltr_Abricot_used;
var float : Couts;
var float : Benefs;

constraint
      forall(i in 1..4) (Tab_litre_Cocktail[i] >= 0.0)
  /\  Couts >= 0.0
  /\  Benefs >= 0.0
  /\  Ltr_Poire_used = Tab_litre_Cocktail[1]*0.5+Tab_litre_Cocktail[2]*0.5
                       +Tab_litre_Cocktail[4]*0.33
  /\  Ltr_Pomme_used = Tab_litre_Cocktail[1]*0.25+Tab_litre_Cocktail[2]*0.5
                       +Tab_litre_Cocktail[3]*0.5+Tab_litre_Cocktail[4]*0.33
  /\  Ltr_Abricot_used = Tab_litre_Cocktail[1]*0.25+Tab_litre_Cocktail[3]*0.5
                       +Tab_litre_Cocktail[4]*0.33
  /\  Ltr_Poire_used <= 20.0
  /\  Ltr_Pomme_used <= 20.0
  /\  Ltr_Abricot_used <= 15.0

  /\  Couts = Ltr_Poire_used*2.5+Ltr_Pomme_used+2.0*Ltr_Abricot_used
              + 3.0*Tab_litre_Cocktail[1]+2.0*Tab_litre_Cocktail[2]+
              2.5*Tab_litre_Cocktail[3]+4.0*Tab_litre_Cocktail[4]
  /\  Benefs = 6.0*Tab_litre_Cocktail[1]+4.5*Tab_litre_Cocktail[2]
               +5.0*Tab_litre_Cocktail[3]+6.5*Tab_litre_Cocktail[4]
              - Couts
  ;
```

Addendum : Primal complet du Pb. de Bernard ...suite

```
solve maximize Benefs;

output ["Benefs= ", show(Benefs), ", Couts = " , show(Couts),
        "\n Litre de chaque Cocktail = " , show(Tab_litre_Cocktail)]
    ++ ["\n Qte utilisée de chaque jus (Poire,Pomme,Abricot) : ",
        show(Ltr_Poire_used), " " ,
        show(Ltr_Pomme_used), " ", show(Ltr_Abricot_used), "\n"];

%=====================================================================
% mzn-g12mip jus-de-fruit.mzn
% Ou
% minizinc -b mip jus-de-fruit.mzn

%  Benefs= 52.5, Couts = 247.5
%  Ltr de chaque Cocktail = [30.0, 10.0, 15.0, 0.0]
%  Qte utilisée de chaque jus (Poire,Pomme,Abricot) : 20.0 20.0 15.0
```

Addendum : Solution CLP à Bernard

```
primal(Benef, Total_couts, Cout_mat_prem_et_benef,
               [Qte_cocktail_1 ,Qte_cocktail_2 ,Qte_cocktail_3 ,Qte_cocktail_4],
     Cout_base_Pr=2.50, Cout_base_Pm=1.00, Cout_base_Ab=2.00
    , Cout_prep_cocktail_1=3, Cout_prep_cocktail_2=2, Cout_prep_cocktail_3=2.5
    , Cout_prep_cocktail_4=4                    %Si 3.5, on n'utilise pas tout

    , Cout_cocktail_1 >=0,  Cout_cocktail_2 >=0,    Cout_cocktail_3 >=0
    , Cout_cocktail_4 >=0

    , Cout_cocktail_1 = 0.5*Cout_base_Pr + 0.25*Cout_base_Pm + 0.25*Cout_base_Ab
                        + Cout_prep_cocktail_1
    , Cout_cocktail_2 = 0.5*Cout_base_Pr + 0.5*Cout_base_Pm +C out_prep_cocktail_2
    , Cout_cocktail_3 = 0.5*Cout_base_Pm + 0.5*Cout_base_Ab+Cout_prep_cocktail_3
    , Cout_cocktail_4 = 0.333*Cout_base_Pr+0.333*Cout_base_Pm+0.333*Cout_base_Ab
                        + Cout_prep_cocktail_4

    , Qte_Pr_dispo= 20, Qte_Pm_dispo= 20, Qte_Ab_dispo= 15

    , Qte_cocktail_1 >=0, Qte_cocktail_2 >=0, Qte_cocktail_3 >=0
    , Qte_cocktail_4 >=0

    , Qte_cocktail_1 =<  0.5*Qte_Pr_dispo  +0.25*Qte_Pm_dispo +0.25*Qte_Ab_dispo
    , Qte_cocktail_2 =<  0.5*Qte_Pr_dispo  +0.5*Qte_Pm_dispo
    , Qte_cocktail_3 =<                     0.5*Qte_Pm_dispo  +0.5*Qte_Ab_dispo
    , Qte_cocktail_4 =<  0.333*Qte_Pr_dispo+0.333*Qte_Pm_dispo+0.333*Qte_Ab_dispo

    , Total_Qte_Pr_used >= 0, Total_Qte_Pr_used =< Qte_Pr_dispo
    , Total_Qte_Pm_used >= 0, Total_Qte_Pm_used =< Qte_Pm_dispo
    , Total_Qte_Ab_used >= 0, Total_Qte_Ab_used =< Qte_Ab_dispo

    , Total_Qte_Pr_used = 0.5*Qte_cocktail_1 +0.5*Qte_cocktail_2+
                          0.333*Qte_cocktail_4
    , Total_Qte_Pm_used = 0.25*Qte_cocktail_1+0.5*Qte_cocktail_2+0.5*Qte_cocktail_3
```

Addendum : Solution CLP à Bernard ...suite

```
                        +0.333*Qte_cocktail_4
    , Total_Qte_Ab_used = 0.25*Qte_cocktail_1+0.5*Qte_cocktail_3+0.5*Qte_cocktail_4

    , Total_couts = Qte_cocktail_1 * Cout_cocktail_1 + Qte_cocktail_2 * Cout_cocktail_2
                   +Qte_cocktail_3 * Cout_cocktail_3 +Qte_cocktail_4 * Cout_cocktail_4

    % Les ventes
    , VC1 = Qte_cocktail_1 * 6
    , VC2 = Qte_cocktail_2 * 4.5
    , VC3 = Qte_cocktail_3 * 5
    , VC4 = Qte_cocktail_4 * 6.5

    , Total_ventes = VC1 +VC2 +VC3 +VC4
    , Benef >= 0
    , Benef = Total_ventes - Total_couts

    , Couts_melange_et_emballage = Qte_cocktail_1 * Cout_prep_cocktail_1
                                 + Qte_cocktail_2 * Cout_prep_cocktail_2
                                 + Qte_cocktail_3 * Cout_prep_cocktail_3
                                 + Qte_cocktail_4 * Cout_prep_cocktail_4

    , Cout_mat_prem_et_benef = Couts_matieres_premieres+ Benef
    },  maximize(Benef).

?- primal(B, T,C, L1,L2).

Benefs = 46.621875
Total couts = 223.3875
Ce qu'il faut donner  = 136.465625
L1 (les qtés des cocktails vendus et donc fabriqués)
        = [18.75, 11.04375, 17.5, 3.125]
L2 (les qtés des jus utilisés) = [15.9375, 20.0, 15.0]

=> Si on met Cout_prep_cocktail_4=4 au lieu de 3,5, on utilisera Tous les jus.
```

Addendum : Solution CLP à Bernard ...suite

```
?-primal(B, T, C, L1, L2).

Benefs = 51.58875
Total couts = 242.82
Ce qu'il faut donner = 151.58875
L1 (les qtés des cocktails vendus et donc fabriqués)  =
         [18.75, 13.7575, 9.375, 11.25]
L2 (les qtés des jus utilisés)  = [20.0, 20.0, 15.0]
```

Remarques :

- Dans la première requête, on n'utilise pas forcément tous les jus.
  ➙ Le cout calculé ne contient pas la valeur du jus (de poire) restant.
- Ce qu'il faut donner au producteur (pour lui acheter tout et qu'il ne nous fasse pas concurrence) :
  ➙ on lui propose sa matière première + les bénéfices.

- Requête 2 : on fixe Cout_prep_cocktail_4 = 4 (au lieu de 3,5),
  ➙ on utilisera Tous les jus.

☞ Faire le dual.

Addendum : un autre exemple de relaxation

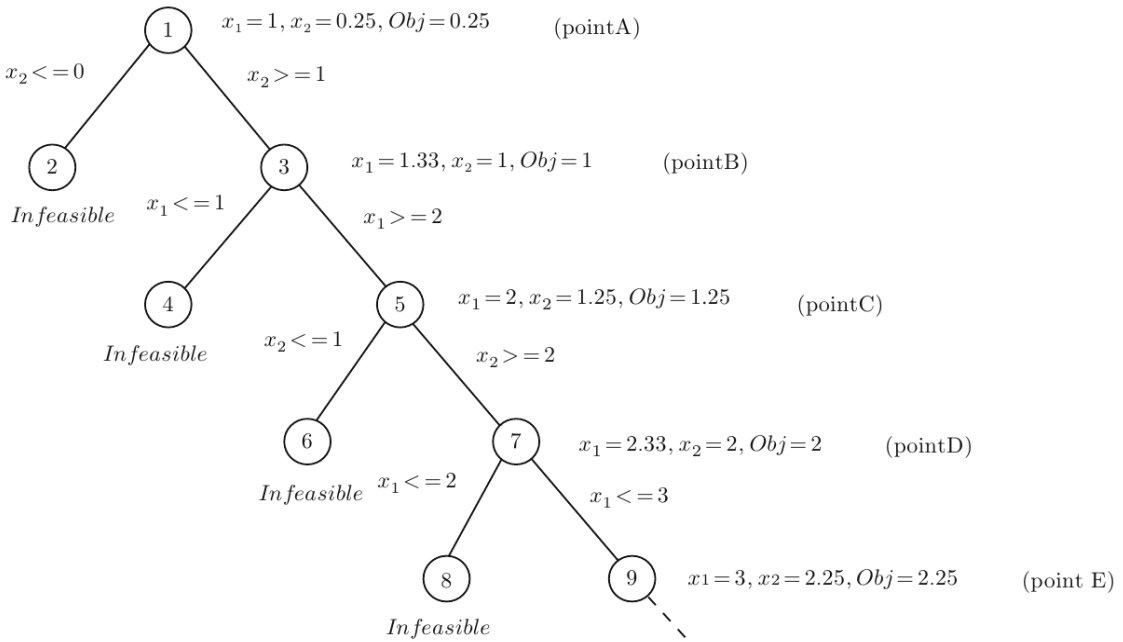
Le problème suivant n'aura pas de solution par B & B.

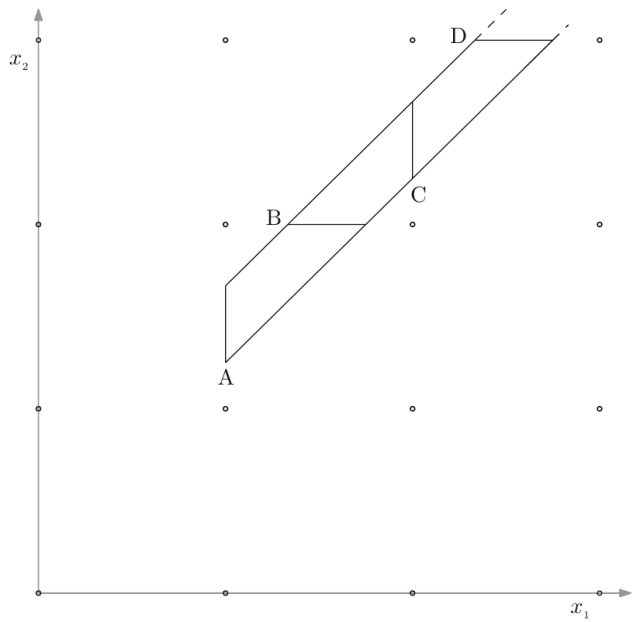
$$
\begin{aligned}
&\textit{maximize} && Z = x2 \\
&\textit{s.t.} && 3x1 - 3x2 \geq 1 \\
& && 4x1 - 4x2 \leq 3 \\
& && x1 \geq 1 \\
&\textit{given} && x1, x2 \in \mathbb{Z}
\end{aligned}
$$

- Voir l'arbre de relaxation et son interprétation géométrique ../..

Addendum : un autre exemple de relaxation ...suite

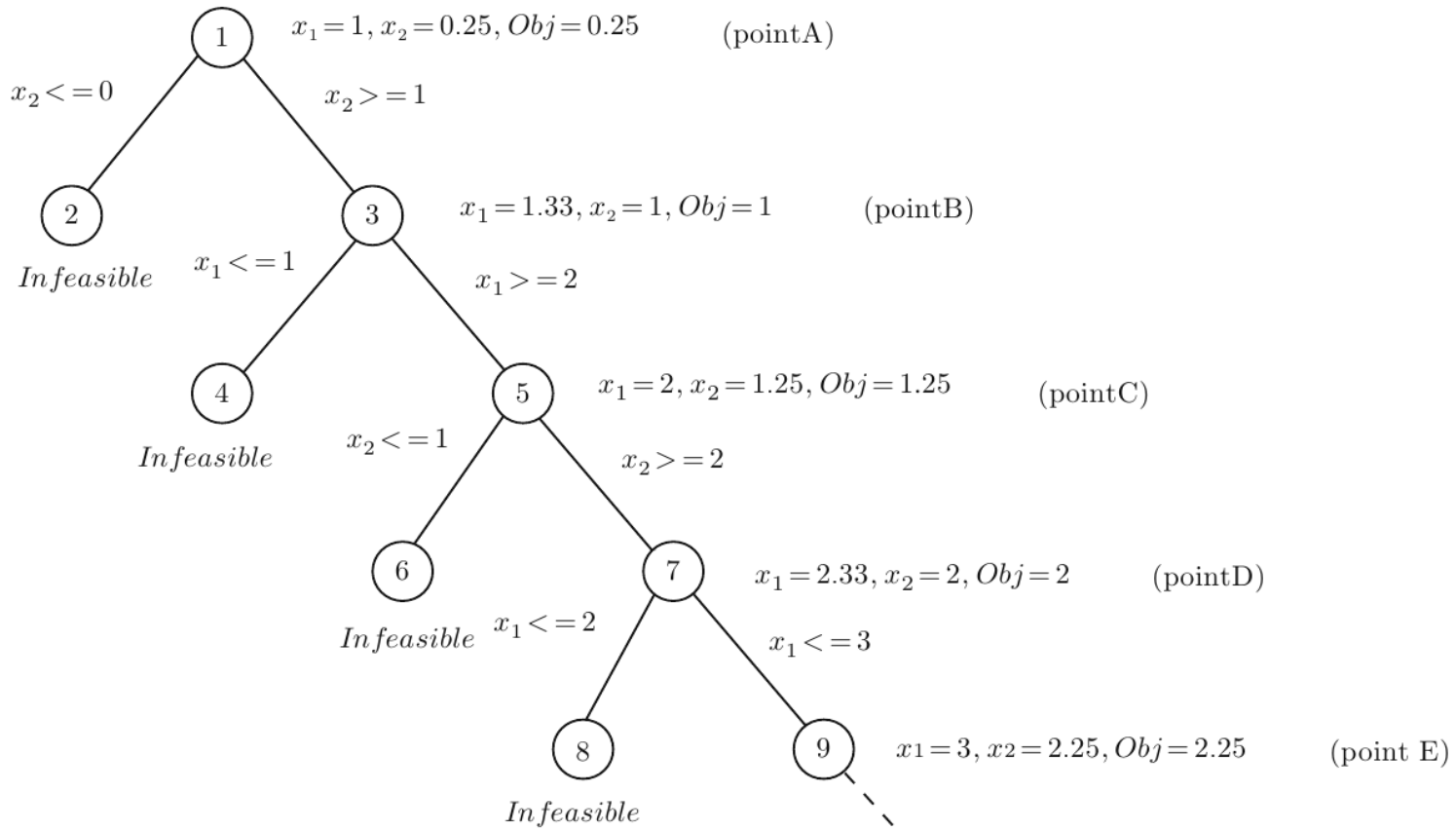
$$
\begin{aligned}
maximize \quad & Z = x2 \\
s.t. \quad & 3x1 - 3x2 \geq 1 \\
& 4x1 - 4x2 \leq 3 \\
& x1 \geq 1 \\
given \quad & x1, x2 \in \mathbb{Z}
\end{aligned}
$$





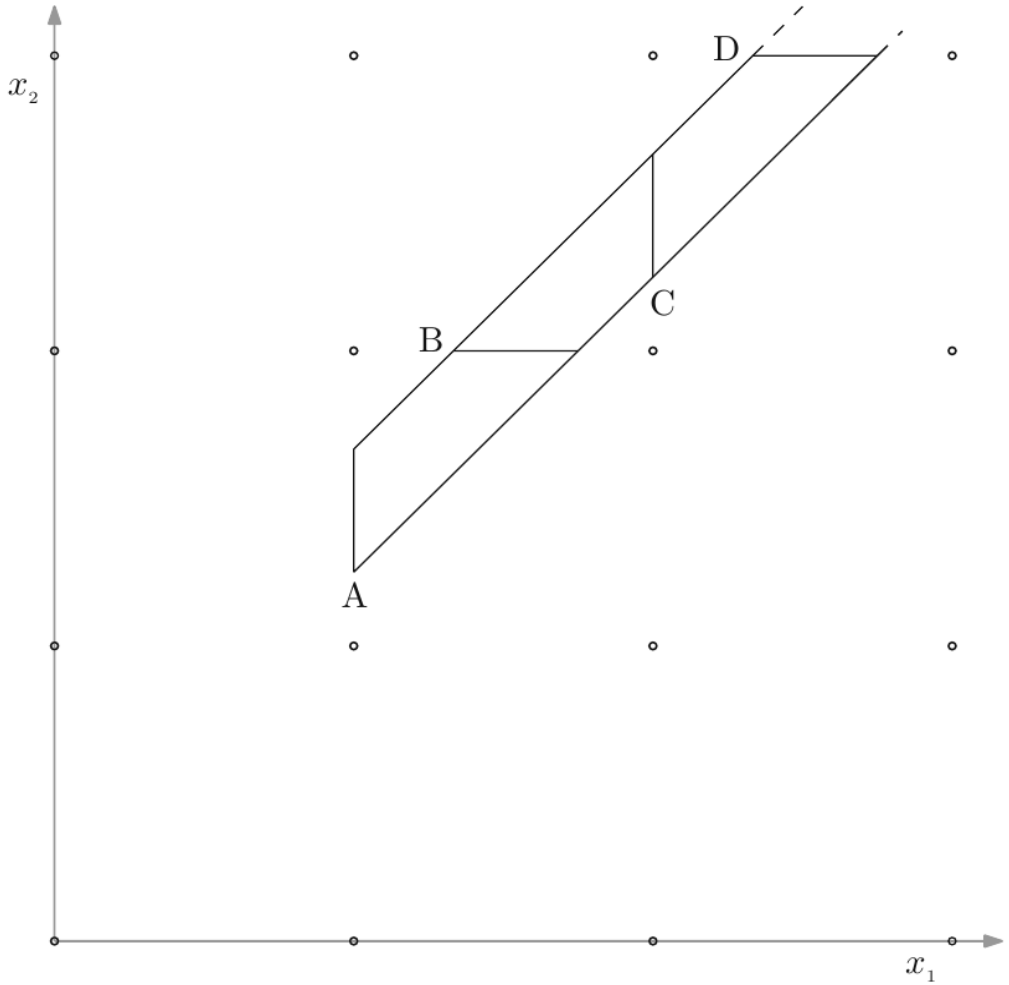
Addendum : un autre exemple de relaxation ...suite

Rappel de l'arbre de relaxation



Addendum : un autre exemple de relaxation ...suite

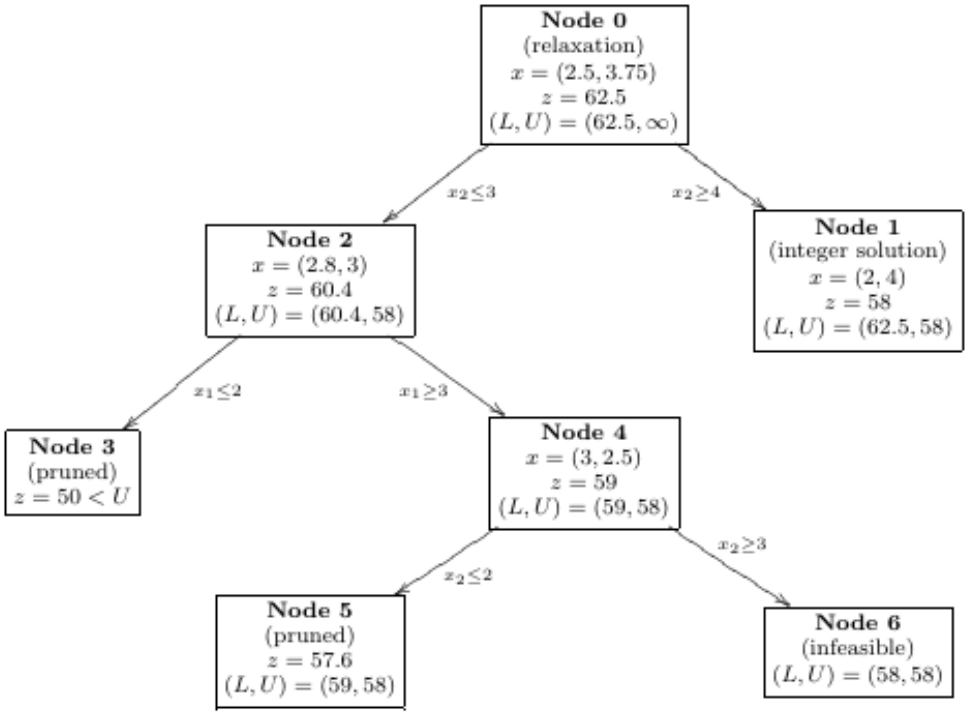
Et son interprétation géométrique



Relaxations B & B : deux autres exemples

Exemple 1 :

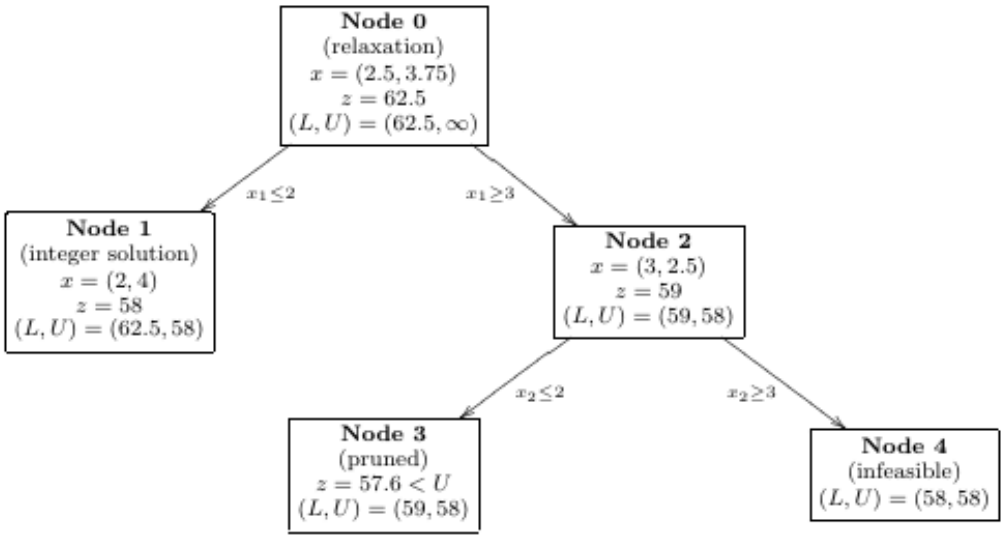
$$\begin{aligned} \max\ z = 13x_1 + 8x_2 \\ x_1 + 2x_2 \leq 10 \\ 5x_1 + 2x_2 \leq 20 \\ x_1 \geq 0, x_2 \geq 0 \\ x_1, x_2 \text{ integer} \end{aligned}$$



Relaxations B & B : deux autres exemples ...suite

Exemple 2 (plus simple) :

$$\begin{aligned} \max\ z = 13x_1 + 8x_2 \\ x_1 + 2x_2 \leq 10 \\ 5x_1 + 2x_2 \leq 20 \\ x_1 \geq 0, x_2 \geq 0 \end{aligned}$$



Addendum B & B : un exemple BIP

• Soit 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	variablede 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.

Modélisation :
• Variables de décision binaires : $\forall j = 1..4, x_j$ *binaire*
• Les contraintes de cout et des bénéfices donnent :
$6x1 + 3x2 + 5x3 + 2x4 \leq 10$ et maximiser $9x1 + 5x2 + 6x3 + 4x4$
• Un seul entrepôt : $x3 + x4 \leq 1$
• Entrepôt si Usine : $x3 \leq x1$ et $x4 \leq x2$

Addendum B & B : un exemple BIP ...suite

- On obtient le système (S)

$$
\begin{aligned}
maximize \quad & Z = 9x1 + 5x2 + 6x3 + 4x4 \\
s.t. \quad & 6x1 + 3x2 + 5x3 + 2x4 \leq 10 \\
& x3 + x4 \leq 1 \\
& -x1 + x3 \leq 0 \\
& -x2 + x4 \leq 0 \\
given \quad & x_j \ binaire \ \forall j = 1..4
\end{aligned}
$$

➙ On applique B&B (à la RO).

1. Branching
2. Bounding
3. Fathoming

Addendum B & B : un exemple BIP ...suite

Etape 1 - Branching : on choisit une variable et on fixe ses valeurs.

- Pour ce cas binaire, on fixe par exemple $x1 = 0$ et $x1 = 1$.
  - Produit deux branches $S1$ et $S0$ descendant de S avec S1 où x1=0 et S0 où x1=1 (x1 disparaît dans S1 et S0, remplacé par 0 ou 1)
  - X1 est appélé *variable de branchement*.
  - On peut choisir judicieusement les variables de branchement mais en général, on choisit dans l'ordre lexicographique.

Etape 2 - Bounding : on *borne* S1 et S0.

- En général, on procède à une **relaxation** des contraintes.
- Ici, on relaxe x_j *binaire* en $x_j \geq 0, x_j \leq 1$
- La relaxation consiste à supprimer (*relâcher*) un sous ensemble des contraintes.
- Dans notre cas, la contrainte difficile est x_j *binaires*.
- En la relâchant, on peut borner les sous-systèmes S1 et S0.

Addendum B & B : un exemple BIP ...suite

- Avec la relaxation, on aura (par Simplex) :
  $$(x1, x2, x3, x4) = (5/6, 1, 0, 1) \text{ avec } Z = 16,5$$
  qui est une borne pour toutes les solutions faisables de S
  - les solutions où les variables sont binaires aura forcément un maximum inférieure à ce $Z = 16,5$.
  - Et puisque les variables sont entiers des (binaires) dans S et leur coefficient sont des entiers, $Z \leq 16$ est dans la solution finale de S.
    On a la limite supérieure pour S : $Z \leq 16$

- Le bounding de S1 donne
  $(x1, x2, x3, x4) = (1, 4/5, 0, 4/5)$ avec $Z \leq 16$
- Pour S0, $(x1, x2, x3, x4) = (0, 1, 0, 1)$ avec $Z \leq 9$
- C'était la 1e itération (y peut y en avoir d'autres).

Addendum B & B : un exemple BIP ...suite

Etape 3 - Fathoming (ou abandon des sous-problèmes):

- Peut avoir lieu de plusieurs façons.

• 1e façon : le fait que la solution de la relaxation de S0 est optimale pour S1 (sans relaxation)

- On a $Z^* = 9$: valeur de référence (référence : *current incumbent*).
- On n'a aucune autre raison de continuer avec S0 puisque tout autre développement de la branche S0 donnera un Z moins bonne.
- On dit que on **élimine** (fathom) S0

• 2e façon : pas besoin de considérer tout autre sous problème avec $bound \leq 9$ car on a déjà S0.

- On dit que tout sous problème est abandonné (*fatomed*) si $Z* \leq 9$
- Ce résultat ne survient pas dans l'itération actuelle car S0 a une borne 16 plus grand que 9 mais il peut survenir dans les autres descendants de S1 avec d'autres branching/relaxations.

Addendum B & B : un exemple BIP ...suite

• 3e facon : si Simplex trouve qu'il n'y a pas de solution faisable avec la relaxation d'un sous problème , alors le sous problème lui même n'a pas de solution faisable et peut être écarté (*fathomed*).

➙ Ce n'est pas le cas ici car on a eu 16 (meilleure que 9).

Résumé :

• Le but des 3 façons a été de savoir si on doit continuer sur les seuls sous-problèmes qui pourraient mener à une meilleure solution, meilleure que le *cout de référence actuelle.*

• Pour résumer : un sous-problème est *éliminé* si

test 1 : sa borne $\leq Z^*$ ou

test 2 : sa LP relaxation n'a pas de solution faisable ou

test 3 : la solution optimale par sa LP relaxation est une valeur entière.

➙ (si sa solution est meilleure que la référence, elle deviendra la nouvelle référence et test-1 est ré-appliqué à tous les sous problèmes non-éliminés avec cette nouvelle meilleure Z^*).

Addendum B & B : un exemple BIP ...suite

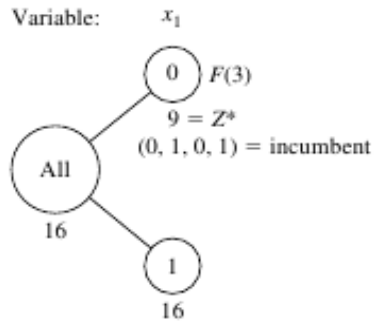
Résumé de la méthode B & B (BIP) :

1. **Branching** : pour tout sous problème non-éliminé, en sélectionner un crée le plus récemment en le subdivisant en deux, en fixant une autre variable de branchement à 0 et 1;
2. **Bounding** : pour tout nouveau sous problème , calculer la borne (*bound*) en appliquant Simplex à sa LP-relaxation et arrondir (par défaut) la valeur de Z ainsi obtenue;
3. **Fathoming** : pour tout nouveau sous problème , appliquer les 3 tests d'élimination et enlever les sous problèmes qui sont écartés par l'un de ces tests.

- **Test d'optimalité** : arrêt s'il n'y a plus de sous problème.
  ➙ La référence est un cout optimal. Sinon, itérer.

Addendum B & B : un exemple BIP ...suite

Suite de l'exemple (on n'avait pas écarté S1).
L'état actuel des calculs est donné par la figure ➙



- L'itération suivante donne $S1.1$ et $S1.0$ à partir de S1 (où $X_1 = 1$).
- Pour S1.0 (avec X1=1 et X2=0) :

$$
\begin{aligned}
&maximize && Z = 9 + 6x3 + 4x4 \\
&s.t. && 5x3 + 2x4 \leq 4 \\
& && x3 + x4 \leq 1 \\
& && x3 \leq 1 \\
& && x4 \leq 0 \\
&given && x_j \ binaire \ \forall j = 3..4
\end{aligned}
$$

➙ Sa LP-relaxation donne $(1, 0, \frac{4}{5}, 0)$ *et* $Z \leq 13\frac{4}{5}$

Addendum B & B : un exemple BIP ...suite

- pour S1.1 (avec X1=1 et X2=1) :

$$
\begin{aligned}
maximize \quad & Z = 14 + 6x3 + 4x4 \\
s.t. \quad & 5x3 + 2x4 \leq 10 \\
& x3 + x4 \leq 1 \\
& x3 \leq 1 \\
& x4 \leq 1 \\
given \quad & x_j \ binaire \ \forall j = 3..4
\end{aligned}
$$

➙ Sa LP-relaxation donne $(1, 1, 0, \frac{1}{2})$ *et* $Z \leq 16$

Les deux sont meilleures que la référence $Z^* = 9$.

Fathoming :

➙ Le test-1 échouera sur S1.1 et S1.0,

➙ Le test-2 aussi car les LP relaxations pour S1.1 et S1.0 ont une solution faisable.

➙ Le test-3 aussi car les deux solutions optimales ont des valeurs non entières.

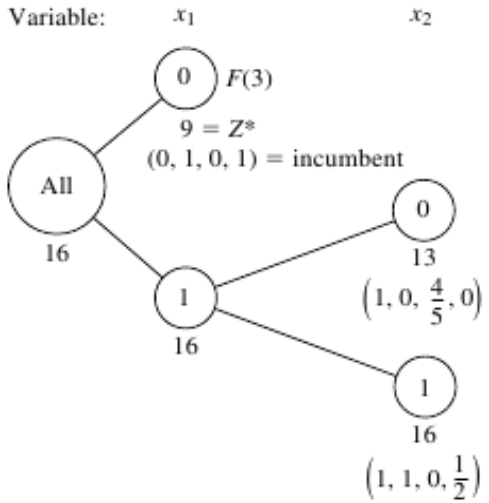
Addendum B & B : un exemple BIP ...suite

- On va à itération 3 .

Addendum B & B : un exemple BIP ...suite

Itération 3 :

Suite de l'exemple (avec S1.1 et S1.0).
L'état actuel des calculs est donné par la figure ➙
N.B. : F(3) : écarté par test 3



• On obtient S1.1.1 et S1.1.0 à partir de S1.1 (voir plus loin pour S1.0).

• S1.1.0 avec X1=1, X2=1, X3=0

$$\begin{aligned} maximize \quad & Z = 14 + 4x4 \\ s.t. \quad & 2x4 \leq 1 \\ & x4 \leq 1 \\ given \quad & x_4 \ binaire \end{aligned}$$

➙ Sa relaxation donne $(1, 1, 0, \frac{1}{2})$ *et* $Z \leq 16$

Addendum B & B : un exemple BIP ...suite

• S1.1.1 avec X1=1, X2=1, X3=1 est de la forme

$$\begin{array}{rr} maximize & Z = 20 + 4x4 \\ s.t. & 2x4 \leq -4 \\ & x4 \leq 0 \\ & x4 \leq 1 \\ given & x_4 \ binaire \end{array}$$

➙ Sa relaxation ne donne pas de solution faisable.

➙ La raison : les contraintes de S1.1.1 sont devenues contradictoires : $x4$ *binaire* contredit $2x4 \leq -4$ et interdit toute relaxation.

➙ S1.1.1 sera éliminé par le test 2.

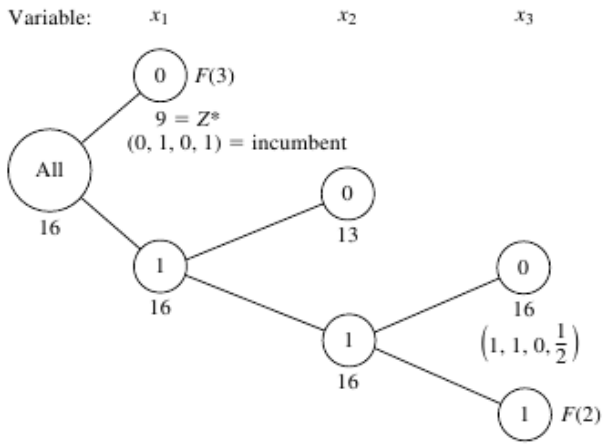
• Pour S1.1.0 : test 2 échoue
ainsi que test 1 (car $16 > 9$);
idem pour test 3 : par la contrainte $2x4 \leq 1$, $x4 = \frac{1}{2}$ n'est pas un entier.

On passe à l'itération 4 sur S1.1.0.

Addendum B & B : un exemple BIP ...suite

Suite de l'exemple après l'itération 4 (à partir de S1.1.0).
L'état actuel des calculs est donné par la figure

- On obtiendra S1.1.0.1 et S1.1.0.0.
- S1.1.0.1 : $x4 = 0 : (1, 1, 0, 0)$ est faisable avec $Z = 14$
- S1.1.0.0 : $x4 = 1 : (1, 1, 0, 1)$ non faisable.



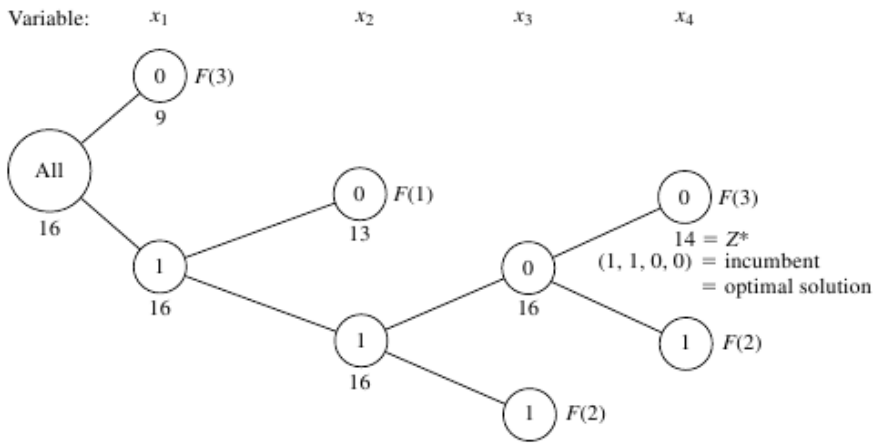
➙ Si on applique les tests d'élimination de manière formelle à S1.1.0.1 et S1.1.0.0, S1.1.0.1 passe test 3 et S1.1.0.0 le test 2.
➙ De plus, S1.1.0.0 est meilleure que la référence $(14 > 9)$
14 devient la référence.
➙ Le fait de passer ces tests permet d'abandonner les développements futurs, cela invalide pas la solution qui a passé le test

- Puisqu'une nouvelle référence est trouvée, on ré-applique test 1 avec la nouvelle valeur de Z^* au seul sous problème S1.0 non encore traité.

Addendum B & B : un exemple BIP ...suite

- Le sous problème S1.0 :
  $bound = 13 \leq Z^* = 14$
  ➙ ce sous problème sera éliminé (raison : test 3).

- L'état final des calculs est donné par la figure ➙



Addendum : détails Plans de coupe

• Une alternative à B & B (pour IP).

Cutting planes (plans de coupe)

◦ Résoudre une relaxation (P_{rel}) linéaire de $P \rightarrow (Z^*, Z^*_{rel})$
➛ Ajouter des plans (géométriques) de coupe quand la solution optimale de la relaxation n'est pas un (nombre) entier,

◦ Il y a des inégalités linéaires $\alpha\ X \leq \alpha_0$
➛ On **coupera** la solution optimale de $\alpha\ X^* > \alpha_0$
➛ On ne rejette aucune solution entière (*valid cut*)

• Proche de Simplex mais des droites (ou plans) viennent délimiter un peu plus la région des solutions, en particulier les solutions non entières.

Plan de coupe

• Utilisée pour résoudre les problèmes de programmation linéaire entière.
• L'idée de base : considérer, à la place d'un IP, sa relaxation linéaire (sans intégralité) et d'exacerber ses inégalités en ajoutant des étapes jusqu'à ce que (idéalement) une solution entière soit trouvée.

• Notation (rappel) : méthode utilisée pour résoudre des programmes entiers (IP) dans la forme (std., normale) :

$$\max\{c^T x \mid Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\}.$$

où A est une matrice réelle et b et c sont des vecteurs de dimension appropriée.

Ici, la condition $Ax \leq b$ s'applique composante par composante, à savoir :

$$a_i \cdot x = \sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

pour toutes les lignes i de la matrice A.

De même, la condition, $x \geq 0$ signifie $\forall j, x_j \leq 0$

Plan de coupe ...suite

Interprétation Géométrique:

le système relaxé (les contraintes d'intégralité omises)

$$P := \{x \mid Ax \leq b, x \geq 0\}, \qquad \text{(relaxation)}$$

forme un **polyèdre convexe** dans un espace à n dimensions qui correspondent aux lignes des hyperplans limitant les inégalités.

• P contient tous les points possibles incluant le système initial, c'est à dire tous les points entiers qui satisfont les contraintes $Ax \leq b$,

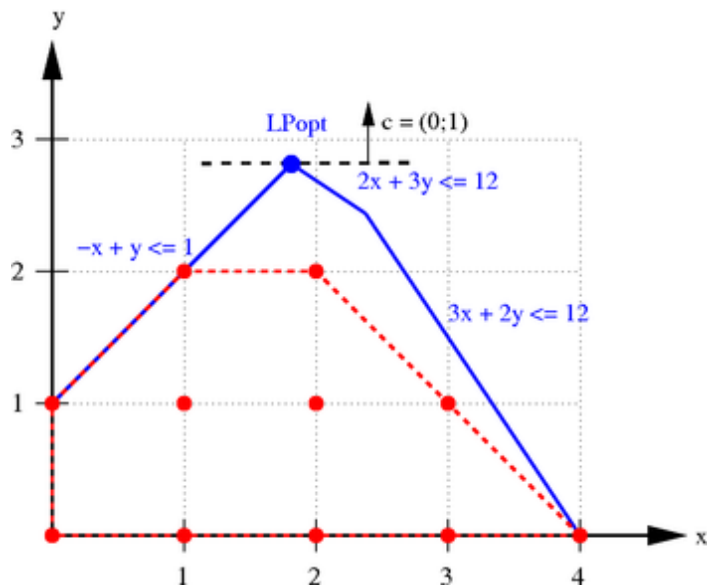
☞ mais tous les points de P ne sont pas recevables (car $\in \mathbb{R}$).

Plan de coupe ...suite

- Exemple :

$$
\begin{array}{lrlll}
\max & y & & & \\
 & -x & +y & \leq 1 \\
 & 3x & +2y & \leq 12 \\
 & 2x & +3y & \leq 12 \\
 & x, y \in \mathbb{Z}_+ & & &
\end{array}
$$

- Les solutions en nombre entier sont en rouge, et les lignes rouges en pointillés indiquent sa **coque convexe**, i.e. le plus petit polyèdre qui contient tous ces points. Ce polyèdre est censé optimal, mais il n'est généralement pas exactement connue.
- Les lignes bleues marquent le polyèdre P de la relaxation LP, qui est donné par les inégalités, sans intégralité.
- La solution optimale du LP vient de la ligne pointillée noire parallèle en partie supérieure (vecteur $c = (0, 1)$).



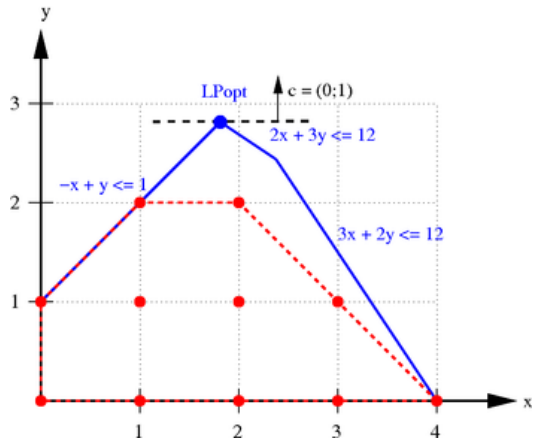
Polytope des points entiers recevables (rouge) avec LP-relaxation (bleu)

Plan de coupe ...suite

Les **solutions** optimales du **problème IP** sont les points entiers $(1;2)$ et $(2;2)$ avec la valeur de la fonction objective (Max)

$$c^T x = (0;1)^T (1;2) = 2.$$

La solution optimale à la LP-relaxation est le point bleu marqués $LP_{opt} = (1,8; 2,8)$, ce qui n'est pas un entier, et donc pas autorisé pour le système (IP) initial.



Polytope des points entiers recevables (rouge) avec LP-relaxation (bleu)

• La méthode "plan de coupe" calcule d'abord une solution à la relaxation linéaire.

• En générale, celle-ci n'est pas un entier (cf. $LP_{opt} = (1.8; 2.8)$), mais fournit une limite supérieure (ici y=2.8)

➙ La valeur optimale du IP (2 dans l'exemple) sera inférieure à la valeur optimale de la relaxation LP (y=2.8 dans l'exemple).

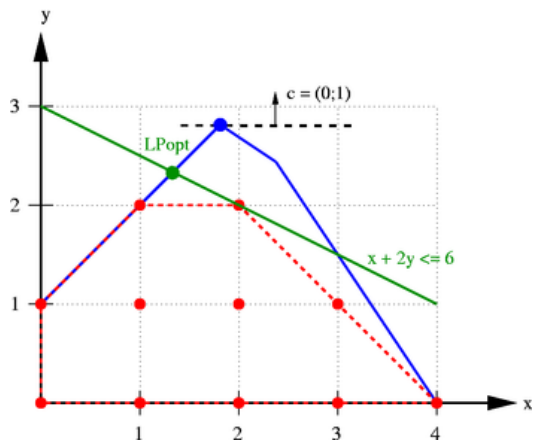
On procède en ajoutant progressivement ces **plans de coupe**.

Plan de coupe ...suite

• Un plan de coupe est une **inégalité** supplémentaire admise par tous les points possibles de l'IP, mais pas par la solution LP actuelle.
➟ L'inégalité est ajoutée au système quand on cherche une autre solution.

• On réitère jusqu'à ce qu'une solution en nombre entier soit trouvée (sera automatiquement optimum pour IP) ou bien plus aucune inégalité est trouvée.

L'interface $x + 2y \leq 6$ (vert) montre que le LP-optimale courant (en bleu) se sépare du IP (en rouge).
➟ Tous les points possibles se trouvent sur un côté de cet hyperplan (en vert), et les solutions au LP de l'autre.
La résolution du système avec cette inégalité supplémentaire donne le point vert LP_{opt} : $(4/3, 7/3)$.



Ajout d'un plan de coupe (en vert)

Plan de coupe ...suite

• LP_{opt} : $(4/3, 7/3)$ n'est pas encore une solution admissible, mais améliore la fonction objectif ($y = 7/3 \approx 2.33$ vs. 2.8).

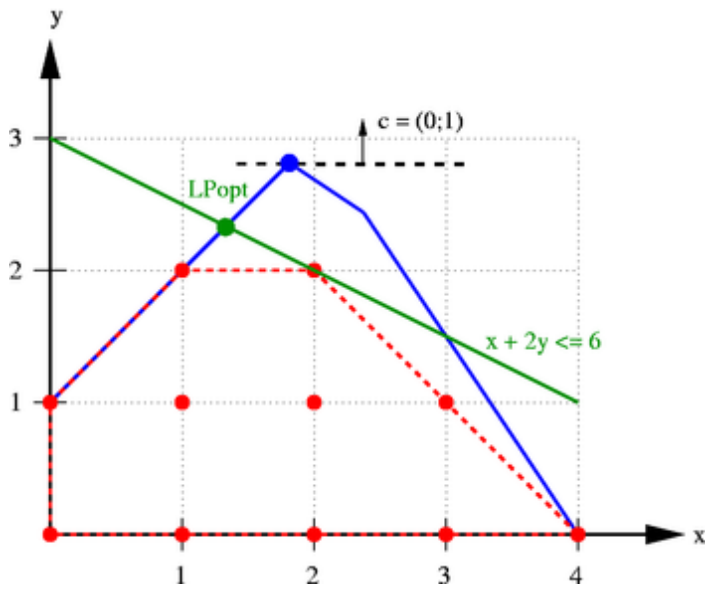
Les meilleurs niveaux de coupe qu'on peut trouver sont donc des ruptures du polyèdre IP avec $(n-1)$ côtés avec n variables.

• Pour l'exemple, les inégalités

$$y \leq 2 \quad \text{et} \quad x + y \leq 4$$

correspondant aux lignes pointillées rouges

.



Ajout d'un plan de coupe (en vert)

Plan de coupe ...suite

Remarques :

• Pour certaines classes d'inégalités, on peut appliquer algorithme suivant:

1- Appliquer et résoudre la relaxation linéaire, soit $\hat{x}$ une solution optimale du LP.

2- Si $\hat{x}$ est un entier, il est également optimale. **STOP**.

3- Tester pour toutes les classes connues d'inégalité, si elles contiennent un ou plusieurs $\hat{x}$ qui ne violent pas les plans de coupe.

4- Si oui, ils s'ajoutent à au système linéaire; passer à 1
Sinon STOP.

• A la fin, si on a trouvé un plan de coupe non violé, sans que la solution LP soit un entier, on peut essayer de déterminer une solution heuristique *entier*, ou on peut entreprendre un *Branch-and-Bound*

➙ Cette combinaison est appelée **Branch-and-cut**.

• Cette technique fonctionne plus ou moins dans la pratique, et selon le problème en main.

Addendum : Génération de Colonne

• Considérons un problème de la découpe de planches de différentes tailles dans une scierie.

➙ Objectif : quelque soit la diversité de la commande, on cherche à réduire les chutes.

• Dans ce type de problèmes combinatoires, le nombre de configurations à générer pour aboutir à une solution minimisant les chutes peut devenir exponentiel.

➙ Peut également utiliser un (très) grand nombre de variables (complexité accrue).

• Dans ces problèmes, on constate que beaucoup de variables seront nulles dans une solution finale :

➙ Des variables (*colonnes* dites patterns / motifs) non utilisées

➙ On a également des configurations (motifs) symétriques (à éliminer).

Addendum : Génération de Colonne ...suite

• Une solution via la technique de Génération de Colonnes est de ne pas se lancer dans la génération de configurations (stratégie Générer / Tester) mais d'utiliser peu de variables au départ (donnant une configuration réduite), puis d'en ajouter au besoin : quand → voir cout réduit ci-dessous.

• On procède donc par la résolution d'une forme réduite de la LP
Puis, si la solution peut être améliorée (indiqué par *cout réduit*), on ajoute une variable choisie (de cout réduit) et on recommence jusqu'à ce que la solution ne puisse plus être améliorée.

• Ajouter une variable → générer une nouvelle configuration
→ générer une nouvelle colonne.

../..

Addendum : Génération de Colonne ...suite

• La génération de colonne utilise la notion du *cout réduit* (*reduced cost*).

• **cout réduit** : la quantité nécessaire dont le coefficient de la fonction objective (le paramètre c ci-dessous) doit *s'améliorer* avant de devenir positive.

➙ On estime de combien la fonction objective changera si un variable (actuellement) nulle devient positive.

• A chaque problème LP, on associe un vecteur de couts réduits :

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

Le **vecteur de couts réduits** associé :

$$\sigma = c - A^T y \qquad \text{où } y \text{ est le vecteur de cout Dual.}$$

☞ Si le *cout réduit* est négatif (pour une nouvelle variable = une nouvelle colonne), alors la solution peut être améliorée en ajoutant cette nouvelle colonne.

Addendum : Génération de Colonne ...suite

Exemple (venant d'un problème de découpe en minimisation des chutes) :
• On a une commande avec $i = 1..k$ lignes où chaque ligne i demande b_i planches d'une certaine longueur,
• Dans la solution envisagée, on considère différentes configurations où
➙ $A_{i,j}$: nbr de fois où une commande i figure dans la configuration j
→ Par exemple, on envisage de découper une planche de 10m et la ligne de commande de planches de 3m y figure 3 fois,
➙ $=x_j$: nbr de fois où la configuration j sera utilisée.

• On aura la système LP :

$$\begin{array}{ll} Min \ \sum_j x_j & x_j \geq 0 \\ \text{s.t.} \ \sum_j A_{ij} \geq b_i. & \end{array}$$

Le vecteur de couts réduits associé :

$$\sigma_j = 1 - \sum_i A_{ji} y_i$$

➙ Si σ_j est négatif, la solution peut être améliorée par x_j

Addendum : Génération de Colonne ...suite

Un autre exemple :

OR: COLUMN GENERATION-EXAMPLE

- Partition a Directed Acyclic Graph into the minimal number of paths.
- *MASTER PROBLEM* ⟹ Choose paths

V: set of variables (n nodes $O(2^n)$)

$x_j \in \{0,1\}$ $x_j = 1$ if Path j is choosen

$\min \Sigma_{xj \in V} x_j$

$\Sigma_{xj \in V} a_{ij} x_j = 1$ each node belongs to one path

⟹ *Provides dual values* λ_i

- Solve the master problem on V' ⊂ V, then add variables by solving the *SUBPROBLEM* ⟹ find a path with negative reduced costs

$y_j \in \{0,1\}$ $y_j = 1$ if node i is on that path

z cost of the path

$z - \Sigma_{i \in V} \lambda_i y_i \leq 0$

⟹ *Provides new columns* x_{ji}

Comparaison OR vs. CSP

- Définition du problème

Modèle CP : CSP
- Vars : $X1, X2, \ldots Xn$
- domaines $D1, D2, \ldots Dn$
- Contraintes : $C(X1, X2, ..., Xk)$
- Objective : $f(X1, X2, ..., Xn)$

Modèle MIP
- min $z = c^T\ x + h^T y$
- s.t. $Ax + By = b$
  $x \geq 0,\ Y \geq 0$
  $x \in \mathbb{Z}, y \in \mathbb{R}$

VARIABLES :
- En CP : sont nommées et ont un domaine de type :
  réel, rationnel, bool, ens, int, symbolique
- En MIP : les variables sont binaires, int, réel (semi continues),
  membre d'un ensemble, non-nulle,

DOMAINES :
- CP : par exemple en CP(FD), le domaine contient des valeurs pouvant êtres assignées aux variables ainsi que celles qui ne sont pas encore démontrées inconsistantes.
- MIP : les variables sont délimitées (avec bornes). L'énumération n'existe pas.

Comparaison OR vs. CSP ...suite

CONTRAINTES :

• CP : ce sont des prédicats (relations) sur un ensemble de variables
  ➙ contraintes de domaine : $X :: [1, 2, 5, 7]$, $Y :: 1..10$
  ➙ contraintes math : $X = Y, X > Y, X \leq Y, \ldots$
  ➙ contraintes symboliques : $\mathit{alldifferent}([X, Y, Z, P])$

• MIP : les contraintes sont des égalités, inégalités entre des termes linéaires ainsi que les contraintes d'intégralité(qui peut être relaxée dans le système LP correspondant).

Contraintes redondantes :

• CP : l'ajout d'une contrainte redondante peut aider l'algorithme de recherche à trouver une solution

• MIP : idem, e.g. des coupure (cuts) valides

☞ Parfois, des contraintes (redondantes) figurent comme des vérifications (a posteriori) des solutions.

Comparaison OR vs. CSP ...suite

Fonction Objectif : même sens dans les deux cas.

Solution faisable (admissible) :

• CP : c'est une assignation variable-valeur qui satisfait toutes les contraintes.

Notion de contrainte-réponse.

L'application d'un e.g. AC/PC peut aussi produire une solution (une seule valeur reste).

➟ Si une telle assignation existe, alors le problème est faisable.

• LP (MIP) : l'espace de solution est donné par l'expression même du système :

$S = \{(x, y) : Ax + By = b, x, y \geq 0\}$ S : ensemble de couples (x, y)

➟ Si S n'est pas vide, le problème est faisable

Comparaison OR vs. CSP ...suite

Modélisation

• CP : formulation plus intuitive, déclarative, flexible
  ➙ intérêt d'utiliser des langages classiques (Imper, fonc, Logiques)

• MP : nécessite plus d'expertise et abstraction (manque d'un moteur algorithmique)
  ➙ Les deux modèles dépendent de celui qui "écrit" !

RELAXATION :

• CP : chaque contrainte est un sous problème indépendant
  ➙ question de faisabilité (partielle donc)
  ➙ L'ajout d'autres contraintes (relaxation) est direct et triviale
• MIP : on relâche quelque contrainte (e.g intégralité)
  ➙ problème d'optimisation

Comparaison OR vs. CSP ...suite

RÉSOLUTION

1- Méthode : recherche dans un arbre

• CP : tout noeud est généré par un *labelling* (cf. CLP)
• A chaque noeud, on a une propagation jusqu'à un point fixe.
  ➙ La propagation de contraintes complète la propriété de consistance
  ➙ L'optimalité est traitée en imposant des contraintes de cout qui propagent (faiblement) aux variables
  ☞ Question des retours arrières (BT).

• MIP : tout noeud est généré en plaçant des bornes aux variables. A chaque noeud, on résout une relaxation linéaire de manière optimale.
  ➙ Si une borne inférieure trouvée est moins bonne que la meilleure actuelle solution, alors le noeud est supprimé (fathomed) car ses successeurs seront pires (e.g. B & B en CSP).

Comparaison OR vs. CSP ...suite

Suite Résolution ...

2- Méthode : propagation de contraintes (seulement CP)

- Au niveau de chaque noeud
- L'algorithme de consistance enlève des valeurs qui ne peuvent pas être dans une solution consistante.
  - ➙ Si un domaine devient vide, alors le problème est infaisable.
- NC,AC , PC, K-consistance, . . .
  - ➙ On peaufine le temps passé au niveau d'un noeud en fonction du nombre de noeuds
  - ➙ Les contraintes peuvent être considérées comme des agents qui activent une propagation à chaque fois qu'un *évènement* arrive (avec éventuellement : modification des bornes/valeurs).

Comparaison OR vs. CSP ...suite

Pré Processing (MIP)

- fixer une valeur à une variable,
- suppression des redondances (vars, col, lignes),
- ajustement des bornes,
- opération de préparation des matrices (rank, norme, ...)

➙ Souvent appliqué au noeud racine, parfois aux autres

Génération de cut (MIP)

➙ Ajout d'inégalités valides

➙ Les cuts globaux sont globalement valides, les cuts locaux valides au niveau de sous arbre.

Comparaison OR vs. CSP ...suite

ORTIMISATION :

• CP : si une solution Z^* est trouvée, on ajoute une contrainte à la fonction objective (représentée par Z) : $Z < Z^*$.

➙ Z étant reliée aux autres variables, il y a propagation et modification des bornes qui à leur tour modifient Z.

• MIP : à chaque noeud, la relaxation est résolue apportant une borne inf. au problème.

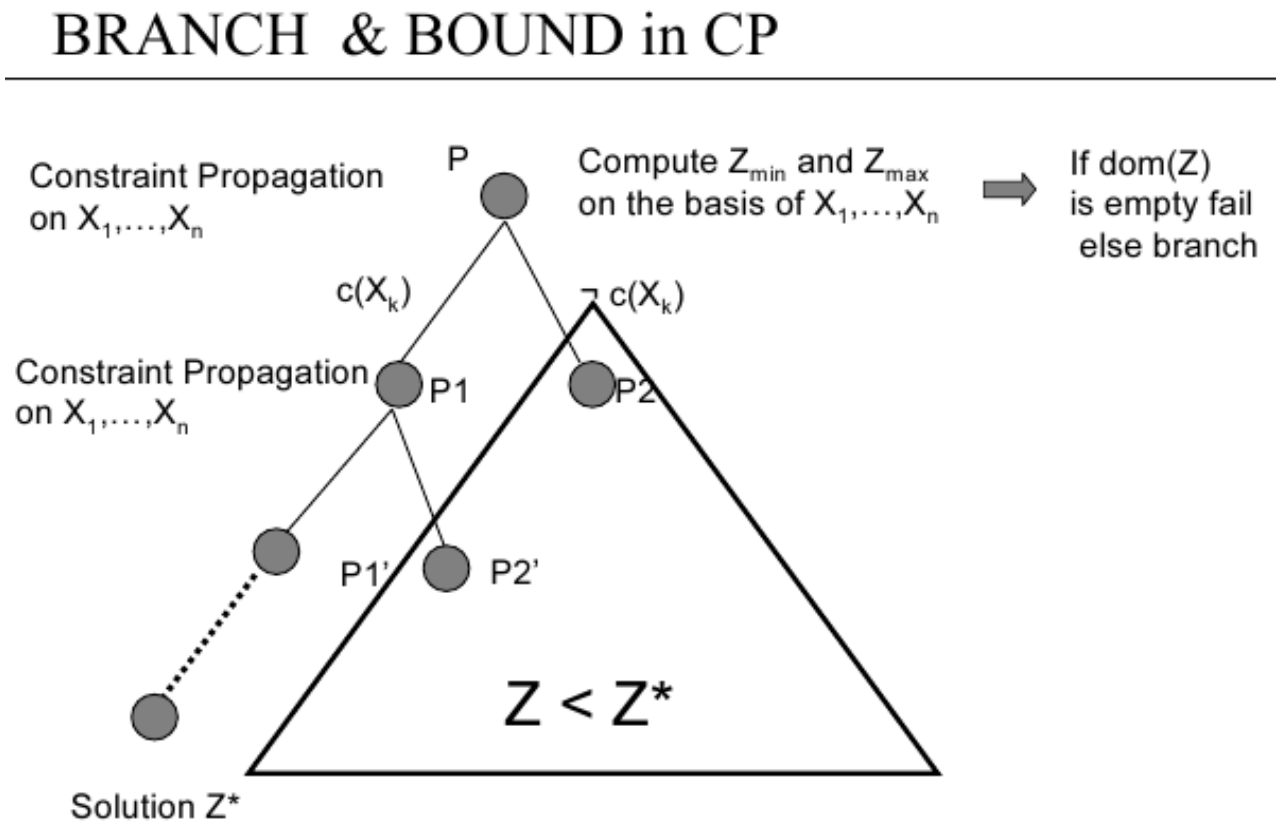
Si la binf est pire que la meilleure bornes sup. actuelle, on élimine le noeud (Fathomed).

Sinon, une variable non entière est sélectionnée et on branche (cf. B&B de LP)...

➙ On calcule en général une bsup. initiale (pour démarrer).

Comparaison OR vs. CSP ...suite

B&B en CP :



Comparaison OR vs. CSP ...suite

Propriétés des problèmes :

• La méthode appliquée doit tenir compte de la taille du problème (espace de recherche).

• Question de symétrie : souvent les problèmes ont des solutions symétriques

➙ CP : on peut imposer des contraintes qui évitent les solutions symétrique et donc la recherche dans la partie symétriques.

➙ En IP : ces contrainte augmentent la taille du problème.

• Techniques heuristiques de branchement selon le problème :

➙ CP : on peut trouver rapidement des solutions

➙ Si on ne peut pas prouver l'optimalité

• L'inconsistance globale est détectable en LP, pas en CP.

• Optimalité/faisabilité :

➙ Dans certains pbs, l'optimalité l'emporte sur la faisabilité ?

➙ e.g. TSP (ou coloration).

➙ La forme de la fonction objective compte.

Tabmat