Programmation par contraintes avec ILOG Solver

Pascal Brisset

Partenariat académique avec ILOG :

- Engagement de l'ENAC :
  - 20h de cours avec le pack optimisation d'ILOG comme support ;
  - citation d'ILOG sur les supports de publicité de l'ENAC.
- Engagement d'ILOG :
  - maintenance gratuite ;
  - promotion de l'ENAC par ILOG au près de ses clients ;
  - information des offres d'emploi requièrant des compétences ILOG ;
  - certification des diplômés de l'ENAC par ILOG.

Plan

- Solver
  - Variables, expressions, contraintes, recherche, optimisation
  - Des solutions pour un exemples
  - Contraintes globales
  - Contraintes définies par l'utilisateur
- Scheduler
  - Tâche, ressource (discrète, unaire, capacitive, énergétique, ...)

1 Mon premier programme avec ILOG Solver

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

- Chaque lettre correspond à un chiffre distinct.
- $M \neq 0$, $S \neq 0$

Codage : 11 variables, 4 équations, 2 inéquations, 1 "tous différents".

```
#include <ilsolver/ilcint.h>
int main(){
 // Initialisation
  IlcManager m(IlcEdit);

 // Déclaration des variables
  IlcIntVar S(m, 0, 9), E(m, 0, 9), N(m, 0, 9), D(m, 0, 9),
            M(m, 0, 9), O(m, 0, 9), R(m, 0, 9), Y(m, 0, 9);
  IlcIntVar send =            1000*S + 100*E + 10*N + D;
  IlcIntVar more =            1000*M + 100*O + 10*R + E;
  IlcIntVar money = 10000*M + 1000*O + 100*N + 10*E + Y;
  IlcIntVarArray letters(m, 8, S, E, N, D, M, O, R, Y);

 // Pose des contraintes
  m.add( S != 0 );
  m.add( M != 0 );
  m.add( send + more == money );
  m.add(IlcAllDiff(letters));
```



```
 // Ajout d'un but d'instanciation des variables
  m.add(IlcGenerate(letters));

 // Recherche d'une solution
  m.nextSolution();

 // Affichage
  m.out() << "    " << send.getValue() << endl
  << " +  " << more.getValue() << endl
  << " = " << money.getValue() << endl;
 // Clôture et libération mémoire
  m.end();
}
```

```
sun10[43]% ./send
ILOG Solver 4.310, licensed to "enac-toulouse"
    9567
 +  1085
 = 10652
```

1.1 Manager

À un problème de satisfaction de contraintes correspond un *manager*. Le *manager* gère les objets (variables, contraintes, ...) et plus généralement l'état du CSP (recherche, ...).

```
class IlcManager {
  public:
    IlcManager(IlcEditMode editMode);
    ostream& out() const;
    void add(IlcConstraint ct) const;
    IlcBool nextSolution() const;
    void end();
}
enum IlcEditMode { IlcNoEdit, IlcEdit };
```

`IlcEdit` : Contrainte seulement " stockée "lors de son ajout.

`IlcNoEdit` : Contrainte " propagée " dès son ajout.

1.2 Variable entière

Une variable entière est attachée à un *manager* et possède un domaine fini initialisé avec un intervalle, un ensemble ou une expression.
Optionnellement, on peut lui associer un nom.

```
class IlcIntVar : public IlcIntExp {
    IlcIntVar(IlcManager m,  IlcInt min, IlcInt max, char* name = 0);
    IlcIntVar(IlcManager m, const IlcIntArray values, char* name = 0);
    IlcIntVar(const IlcIntExp exp);
}
```

Le domaine d'une variable peut être modifié par suppression de valeurs et par instanciation : ce sont les contraintes qui effectuent ces modifications.

On pourra dire *inconnue* pour la distinguer d'une variable de C++.

1.3 Tableau de variables entières

ILOG Solver propose des classes tableaux distinctes des tableaux natifs du langage. Un tableau Solver contient sa taille.

```
class IlcIntVarArray {
 public:
  IlcIntVarArray(IlcManager m,IlcInt size,const IlcIntVar exp0,exp1 ...);
  IlcIntVarArray(IlcManager m,IlcInt size);
  IlcIntVarArray(IlcManager m,IlcInt size,IlcInt min,IlcInt max);
  IlcInt getSize() const;
}
```


1.4 Expression entière

Une expression entière est constituée de variables entières et d'entiers et construite avec les opérateurs arithmétique usuels.

```
IlcIntExp operator+ (IlcIntExp exp1, IlcInt exp2);
IlcIntExp operator+ (IlcInt exp1, IlcIntExp exp2);
IlcIntExp operator+ (IlcIntExp exp1, IlcIntExp exp2);
```

Une expression possède un domaine qui n'est pas représenté mais calculé à partir des constituants.

```
class IlcIntExp {
    IlcInt getValue() const;
    IlcInt getSize () const;
    ...
}
```

La classe `IlcIntVar` hérite de la classe `IlcIntExp`.

1.5 Contrainte

Une contrainte est une relation qui porte sur une ou plusieurs variables. La plupart des opérateurs prédicatifs permettent de construire des contraintes.

```
IlcConstraint operator == (IlcIntExp    exp1, IlcIntExp    exp2);
IlcConstraint operator == (IlcIntExp    exp1, IlcInt       exp2);
IlcConstraint operator == (IlcInt       exp1, IlcIntExp    exp2);
```

Une contrainte doit être ajoutée (action *tell*) au *manager* avec la méthode `add`.

De nombreuses contraintes *globales* sont disponibles :

```
IlcConstraint IlcAllDiff(const IlcAnyVarArray array,
                         IlcWhenEvent event=IlcWhenValue);
enum IlcWhenEvent { IlcWhenValue, IlcWhenRange, IlcWhenDomain};
```

Condition de propagation de la contrainte :

IlcWhenValue une des variables est instanciée ;

IlcWhenRange une borne d'une des variables est modifiée ;

IlcWhenDomain le domaine d'une des variables est modifié.

1.6 But

La recherche d'une solution est faite avec un but, comme en Prolog. Un certain nombre de prédicats prédéfinis peuvent être composés par conjonction et disjonction.

```
IlcGoal IlcGenerate(const IlcIntVarArray,
      IlcChooseIntIndex chooseVariable=IlcChooseFirstUnboundInt);
typedef IlcInt (*IlcChooseIntIndex) (const IlcIntVarArray);
```

Le but `IlcGenerate` sélectionne les variables dans l'ordre spécifié (`IlcChooseIntIndex`) et tente de les instancier par ordre croissant des valeurs du domaine.

Un but doit être ajouté au *manager* avec la méthode `add`.

La classe `IlcConstraint` hérite de la classe `IlcGoal`.

1.7 Recherche, affichage et terminaison

La méthode `nextSolution()` du *manager* lance la résolution des buts précédemment ajoutés. Le booléen `IlcTrue` est retourné si une solution est trouvée.

La méthode `out()` du *manager* fournit un canal de trace associé (par défaut `cout`) qui peut être défini avec `openLogFile`.

La méthode `end()` du *manager* doit être appelée pour terminer proprement le calcul et libérer la mémoire associée.

1.8 Structure de PPC avec ILOG Solver

L'enchaînement suivant sera toujours respecté:

1. création d'un *manager* ;
2. définition des variable ;
3. pose des contraintes ;
4. création des buts d'étiquetage;
5. recherche d'une (des) solution(s) ;
6. éventuellement: optimisation de la solution;
7. affichage de la (des) solution(s) ;
8. clôture du *manager*.

2 Manager

Le *manager* est le premier objet qu'il faut construire et le dernier à détruire (méthode `end()`).

2.1 Création

```
IlcManager(IlcEditMode editMode);
```

Deux modes possibles à la création :

IlcEdit Les contrainte sont seulement stockées et *suspendues* jusqu'à la recherche de solution. Une contrainte peut également alors être supprimée.

IlcNoEdit Les contraintes sont *propagées* immédiatement

Le mode `IlcEdit` est supprimé avec `nextSolution` et peut être rétabli avec `restart`.

2.2 Ajout de contraintes et buts

Le *manager* implémente une *constraint queue*, une collection de contraintes gérée avec les méthodes

```
void add(IlcConstraint ct) const;
void remove(IlcConstraint ct) const;
```

Pour la propagation, le *manager* utilise une liste de variables *modifiées* dont les contraintes doivent être vérifiées (établissement de l'arc-consistance).

Le *manager* implémente une file de buts (la résolvante de Prolog) gérée avec les méthodes

```
void add(IlcGoal goal) const;
void remove(IlcGoal goal) const;
```

dont l'effet dépend du mode courant.

Les buts sont exécutés dans l'ordre d'ajout au *manager*.

2.3 Résolution

La méthode `nextSolution()`

- passe du mode *edit* à la recherche ;
- *post* toutes les contraintes ;
- empile tous les buts sur la pile ;
- lance la résolution des buts.

La gestion des buts est équivalente à celle de Prolog : buts sélectionnés dans l'ordre et points de choix.

La valeur retournée est `IlcTrue` si une solution est trouvée, `IlcFalse` sinon. Un second appel après un succès continue la recherche là où elle avait était stoppée (`;` de Prolog).

Exemple : calcul de toutes les solutions.

```
while(m.nextSolution()) { /* Affichage de la solution */ }
```

2.4 Optimisation

Exemple: rendre la monnaie avec des pièces de 1, 2, 5, 10 et 20 avec le moins de pièces possible.

```
#include <ilsolver/ilcint.h>
main() {
  IlcManager m(IlcEdit);
 // Données
  IlcInt somme = 123;
  IlcIntArray valeurs(m, 5, 1, 2, 5, 10, 20);
 // Variables
  IlcIntVarArray nb_pieces(m, 5, 0, somme);
  IlcIntVar cout(m, 0, somme);
 // Contraintes
  m.add(IlcScalProd(valeurs, nb_pieces) == somme);
  m.add(IlcSum(nb_pieces) == cout);
 // But
```



```
  m.add(IlcGenerate(nb_pieces));

 // Critère d'optimisation
  m.setObjMin(cout);

 // Recherche de la meilleure solution
  while(m.nextSolution()) {
    m.out() << nb_pieces << " : " << cout << endl;
  }
  m.end();
}
```

```
sun10[42]% ./coins
ILOG Solver 4.310, licensed to "enac-toulouse"
IlcIntVarArrayI[[0] [4] [1] [1] [5]] : [11]
IlcIntVarArrayI[[1] [1] [0] [0] [6]] : [8]
```

La méthode du *manager*

```
void setObjMin(IlcIntVar obj, IlcInt step = 1);
```

permet de fixer un objectif de minimisation: cela contraint la recherche à produire une solution meilleure d'au moins `step` que la dernière solution trouvée.

NB : la variable de coût doit être instanciée lors de la recherche.

Pour une maximisation, utiliser `m.setObjMin(-cout)`.

Autre problème: le moins de pièces possibles pour pouvoir toujours rendre la monnaie: contraintes précédentes pour tout $i = 1..100$.


```
#include <ilsolver/ilcint.h>
main() {
  IlcManager m(IlcEdit);
  IlcInt somme_max = 100;
  IlcIntArray valeurs(m, 5, 1, 2, 5, 10, 20);
  IlcIntVarArray nb_min_pieces(m, 5, 0, somme_max);
  IlcIntVar cout(m, 0, somme_max);

  for(int i = 1; i <= somme_max; i++) {
    IlcIntVarArray nb_pieces(m, 5, 0, somme_max);

    m.add(IlcScalProd(valeurs, nb_pieces) == i);
    for(int j = 0; j < 5; j++) {
      m.add(nb_pieces[j] <= nb_min_pieces[j]);
    }
    m.add(IlcGenerate(nb_pieces));
  }
```

```
  m.add(IlcSum(nb_min_pieces) == cout);

  m.add(IlcGenerate(nb_min_pieces));

  m.setObjMin(cout);
  nb_min_pieces.setStorable();
  while(m.nextSolution()) {
    m.out() << "best so far : " << cout << endl;
    m.storeSolution();
  }
  m.restart();
  m.nextSolution();
  m.out() << nb_min_pieces << " : " << cout << endl;

  m.printInformation();
  m.end();
}
```



```
sun10[35]% ./coins2
ILOG Solver 4.310, licensed to "enac-toulouse"
best so far : [12]
best so far : [11]
best so far : [10]
best so far : [9]
IlcIntVarArrayI[[1] [2] [1] [1] [4]] : [9]
```

La méthode `restart()` restaure l'état du *manager* tel qu'il était avant le premier appel à `m.nextSolution()`. La meilleur valeur trouvée pour `setObjMin` est conservée.

NB: Les contraintes et les buts étants gérés indépendamment (*constraint store* et *goal stack*), les `m.add` des deux types peuvent être entrelacés.

2.5 Mémorisation d'une solution

Une variable peut être notée *mémorisable* avec la méthode `setStorable()` des classes `IlcIntVar` et `IlcIntVarArray`.

La méthode `storeSolution()` du *manager* permet de mémoriser la valeur courante de toutes les variables marquées mémorisables.

La méthode `restart()` instancie les variables marqués avec les valeurs mémorisées: l'appel suivant à `nextSolution()` retrouve immédiatement la solution.

```
...
  nb_min_pieces.setStorable();
  while(m.nextSolution()) {
    m.out() << "best so far : " << cout << endl;
    m.storeSolution();
  }
...
```


2.6 Informations sur le *manager*

À tout moment la méthode `printInformation()` peut être appelée pour afficher des informations concernant l'état du *manager*.

```
Number of fails               : 2394
Number of choice points       : 2416
Number of variables           : 506
Number of constraints         : 601
Reversible stack (bytes)      : 36204
Solver heap (bytes)           : 241224
Solver global heap (bytes)    : 4044
And stack (bytes)             : 8064
Or stack (bytes)              : 16104
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 6116
Total memory used (bytes)     : 315800
Running time since creation   : 2.82
```

3 Trois modèles pour huit reines

Problème : placer 8 (n) reines sur un échiquier sans qu'aucune n'en menace une autre, i.e. sans que deux reines soient sur une même horizontale, verticale ou diagonale.

Différentes modélisations :

- différentes contraintes ;
- différentes stratégies de recherche ;
- différentes efficacités ;
- différentes tailles de problèmes solubles.

3.1 Des booléens

Variables : $n \times n$ variables booléennes

Contraintes :

- seulement n reines ;
- pas de prise sur les lignes, colonnes et diagonales.

```
#include <ilsolver/ilcint.h>
main(int argc, char **argv) {
  IlcManager m(IlcEdit);
  int n = atoi(argv[1]);  // Taille échiquier = argument commande

  IlcIntVarArray vars(m, n*n, 0, 1);

  m.add(IlcSum(vars) == n);
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
```

```
      for(int k = 1; i+k < n; k++)
        m.add(vars[i*n+j]+vars[(i+k)*n+j] < 2);
      for(k = 1; j+k < n; k++)
        m.add(vars[i*n+j]+vars[i*n+(j+k)] < 2);
      for(k = 1; i+k < n && j+k < n; k++)
        m.add(vars[i*n+j]+vars[(i+k)*n+(j+k)] < 2);
      for(k = 1; i+k < n && j-k >= 0; k++)
        m.add(vars[i*n+j]+vars[(i+k)*n+(j-k)] < 2);
    }
  }

  m.add(IlcGenerate(vars));
  m.nextSolution();
  m.out() << vars << endl;
  m.end();
}
```


Résultat : beaucoup de variables, de contraintes, et d'échecs

```
Number of fails               : 8540
Number of choice points       : 8551
Number of variables           : 64
Number of constraints         : 729
Running time since creation   : 2.35
```

3.2 Des couples

Dans le modèle précédent, peu de variables prennent la valeur vrai ; il y beaucoup plus de cases vides que de cases occupées.

L'idée du second modèle est de ne représenter que les positions des reines :

- $2 \times n$ variables ;
- pas de prise sur les lignes, les colonnes et diagonales.

```
...
  IlcIntVarArray xs(m, n, 0, n-1);
  IlcIntVarArray ys(m, n, 0, n-1);
  for(int i = 0; i < n; i++) { for(int j = i+1; j < n; j++) {
      m.add(xs[i] != xs[j]); m.add(ys[i] != ys[j]);
      m.add(xs[j]-xs[i] != ys[j]-ys[i]);
      m.add(xs[i]-xs[j] != ys[j]-ys[i]);
  } }
  m.add(IlcGenerate(xs)); m.add(IlcGenerate(ys));
```


Beaucoup moins d'espace et 100 fois plus rapide :

```
Number of fails              : 223
Number of choice points      : 235
Number of variables          : 16
Number of constraints        : 112
Running time since creation  : 0.03
```

3.3 Des entiers

Les résultats du modèle précédent

```
sun10[36]% ./queens2 8
ILOG Solver 4.310, licensed to "enac-toulouse"
IlcIntVarArrayI[[0] [1] [2] [3] [4] [5] [6] [7]]
IlcIntVarArrayI[[0] [4] [7] [5] [2] [6] [1] [3]]
```

suggèrent de ne positionner qu'une reine par ligne :

- n variables ;
- pas de prise sur les colonnes et diagonales.


```
  IlcIntVarArray ys(m, n, 0, n-1);
  for(int i = 0; i < n; i++) {
    for(int j = i+1; j < n; j++) {
      m.add(ys[i] != ys[j]);
      m.add(j-i != ys[j]-ys[i]);
      m.add(i-j != ys[j]-ys[i]);
  } }
  m.add(IlcGenerate(ys));
...
```

Espace de recherche 10 fois moins grand.

```
Number of fails               : 24
Number of choice points       : 26
Number of variables           : 8
Number of constraints         : 84
Running time since creation   : 0.01
```

3.4 Contrainte globale

Les contraintes de non-prise sur les colonnes exprime que tous les y's sont distincts. Une contrainte *globale* peut les remplacer.

```
m.add(IlcAllDiff(ys));
for(int i = 0; i < n; i++) { for(int j = i+1; j < n; j++) {
    m.add(j-i != ys[j]-ys[i]);
    m.add(i-j != ys[j]-ys[i]);
} }
```

La strucure de l'espace de recherche n'est pas modifiée ; moins de contraintes et efficacité comparable (resp. 22s et 19s pour 20 reines).

```
Number of fails               : 24
Number of choice points       : 26
Number of variables           : 8
Number of constraints         : 57
Running time since creation   : 0
```


3.5 Moins de contraintes, plus de variables

En remarquant les équivalences suivantes

$$\begin{aligned} j - i \neq ys[j] - ys[i] &\rightarrow ys[i] - i \neq ys[j] - j \\ i - j \neq ys[j] - ys[i] &\rightarrow ys[i] + j \neq ys[j] + j \end{aligned}$$

l'utilisation de variables supplémentaires permet de simplifier encore l'expression des contraintes.

```
...
  IlcIntVarArray ys(m, n, 0, n-1), ys1(m, n), ys2(m, n);
  for(int i = 0; i < n; i++) {
    ys1[i] = ys[i] - i; ys2[i] = ys[i] + i;
  }
  m.add(IlcAllDiff(ys));
  m.add(IlcAllDiff(ys1));
  m.add(IlcAllDiff(ys2));
...
```

Une contrainte *globale propage* en général plus qu'un ensemble de contraintes équivalent : $x_{\{1,2\}} \neq y_{\{1,2\}}$, $x_{\{1,2\}} \neq z_{\{1,2\}}$, $y_{\{1,2\}} \neq z_{\{1,2\}}$.

Moins de contraintes ; performances améliorées (resp. 19s et 10s pour 20 reines).

```
Number of fails              : 24
Number of choice points      : 26
Number of variables          : 22
Number of constraints        : 17
Running time since creation  : 0
```


3.6 Stratégie de recherche

Deux choix pour la recherche :

- l'ordre des variables à instancier ;
- l'ordre des valeurs à tenter.

La stratégie *standard* consiste à choisir la première variable non instanciée et à tenter les valeurs du domaine dans l'ordre croissant.

Pour 20 reines :

```
Number of fails              : 37320
Number of choice points      : 37330
Number of variables          : 98
Number of constraints        : 41
Running time since creation  : 10.27
```

3.6.1 "First Fail"

Une stratégie classique consiste à choisir la variable de plus petit domaine en priorité.

Le second argument de `IlcGenerate` permet de spécifier l'ordre d'instanciation. De nombreux ordres sont prédéfinis :

`IlcChooseFirstUnboundInt` choix par défaut

`IlcChooseMaxMaxInt` variable de plus grand maximum

`IlcChooseMaxMinInt` variable de plus grand minimum

`IlcChooseMaxSizeInt` variable de plus grand domaine

`IlcChooseMinMaxInt` variable de plus petit maximum

`IlcChooseMinMinInt` variable de plus petit minimum

`IlcChooseMinSizeInt` variable de plus petit domaine


```
...
m.add(IlcGenerate(ys, IlcChooseMinSizeInt));
...
```

Pour le problème des reines, la recherche est considérablement améliorée. Pour 20 reines :

```
Number of fails              : 33
Number of choice points      : 43
Number of variables          : 98
Number of constraints        : 41
Running time since creation  : 0.01
```

3.6.2 Encore mieux pour les reines

Choix en priorité de la variable de plus petit domaine et de plus petit minimum.

Le second argument de `IlcGenerate` est en fait une fonction qui doit retourner un index.

```
IlcGoal IlcGenerate(const IlcIntVarArray,
         IlcChooseIntIndex chooseVariable=IlcChooseFirstUnboundInt);
typedef IlcInt (*IlcChooseIntIndex) (const IlcIntVarArray);
```

Les macros `IlcChooseIndex1` et `IlcChooseIndex2` permettent de fabriquer une telle fonction avec 1 ou 2 critères : le critère est un entier caractérisant LA variable `var` [a]. La variable choisie est celle pour laquelle le critère est minimal.

```
IlcChooseIndex2(name, criterion1, criterion2, varType);
```

[a] Sans commentaires !


```
IlcChooseIndex2(IlcChooseMinSizeMin,
                var.getSize(),
                var.getMin(),
                IlcIntVar)
```

Toujours pour 20 reines :

```
Number of fails              : 25
Number of choice points      : 39
Number of variables          : 98
Number of constraints        : 41
Running time since creation  : 0.02
```

Et pour mille et une reines :

```
Number of fails              : 0
Number of choice points      : 500
Number of variables          : 3001
Number of constraints        : 2003
Running time since creation  : 15.04
```

3.6.3 Ordre sur les valeurs tentées

L'ordre de d'essai des valeurs du domaine peut être spécifié avec un objet de la classe `IlcIntSelect`.

Un constructeur permet fabriquer l'heuristique qui minimise un critère appliqué à chaque valeur du domaine : pour tenter en priorité les grandes valeurs, on écrit :

```
IlcInt eval_max(IlcInt val, IlcIntVar var) { return -val; }

main() {
  IlcManager m(IlcEdit);
  ...
  IlcIntSelect selector(m, eval_max);
  ...
  m.add(IlcGenerate(ys, IlcChooseMinSizeInt, selector));
  ...
}
```


L'autre solution consiste à implémenter une sous-classe de `IlcIntSelectI` en définissant la méthode `select`.

```
class IlcIntSelectMaxI : public IlcIntSelectI {
public:
  IlcIntSelectMaxI(){}
  IlcInt select(IlcIntVar var) { return var.getMax(); }
};
IlcIntSelect SelectMax(IlcManager m) {
  return new (m.getHeap()) IlcIntSelectMaxI();
}

main() {
  ...
  IlcIntSelect select = SelectMax(m);
  m.add(IlcGenerate(ys, IlcChooseMinSizeInt, select));
  ...
}
```

4 Contraintes globales

4.1 IlcAllDiff

```
IlcConstraint IlcAllDiff(const IlcAnyVarArray array,
                         IlcWhenEvent event=IlcWhenValue);
enum IlcWhenEvent { IlcWhenValue, IlcWhenRange, IlcWhenDomain};
```

Condition de propagation de la contrainte :

IlcWhenValue une des variables est instanciée ;

IlcWhenRange une borne d'une des variables est modifiée ;

IlcWhenDomain le domaine d'une des variables est modifié.

4.2 IlcInverse

```
IlcConstraint IlcInverse(IlcIntVarArray f, IlcIntVarArray invf);
```

Équivalent à une définition avec des contraintes élémentaires :

```
IlcConstraint Inverse(IlcIntVarArray f, IlcIntVarArray invf) {
  IlcInt n = f.getSize();
  IlcInt m = invf.getSize();
  IlcConstraint c = (0 == IlcIntVar(f.getManager(), 0, 0)); // Vrai

  for(IlcInt i = 0; i < n; i++) {
    c = c && (f[i] >= 0 && f[i] < m) <= (invf[f[i]] == i);
  }
  for(IlcInt j = 0; j < m; j++) {
    c = c && (invf[j] >= 0 && invf[j] < n) <= (f[invf[j]] == j);
  }
  return c;
}
```

4.3 IlcDistribute

```
IlcConstraint IlcDistribute(IlcIntVarArray cards,
                            IlcIntArray    values,
                            IlcIntVarArray vars,
                            IlcFilterLevel level=IlcBasic);
enum IlcFilterLevel { IlcBasic, IlcExtended };
```

La contrainte exprime que dans le tableau `vars`, la valeur `values[i]` apparaît `cards[i]` fois (les tableaux `cards` et `values` doivent être de même taille).

Avec `IlcBasic`, on obtient la même propagation mais une efficacité supérieure à une version *à la main* :

```
IlcIndex j(m);
IlcInt size = cards.getSize();
for (IlcInt i = 0; i < size; i++)
  m.add(cards[i] == IlcCard(j, vars[j] == values[i]));
```

où `IlcCard` est une expression portant sur une *contrainte générique* permettant de spécifier le nombre d'éléments d'un tableau vérifiant une propriété (une contrainte) :

```
IlcIntExp IlcCard(IlcIndex& i, IlcConstraint ct);
```

Un index est un objet syntaxique permettant d'abstraire une contrainte par rapport à un élément de tableau.

```
class IlcIndex { public: IlcIndex(IlcManager m); }
```

4.3.1 Séquence magique

Une *séquence magique* est une séquence $x_0, x_1, ..., x_n$ de nombres telle que 0 apparaît x_0 fois dans la séquence, 1 apparaît x_1 fois, ..., i apparaît x_i fois.

```
IlcIntVarArray vars(m, n+1, 0, n+1);
IlcIntArray coeffs(m, n+1);

for (IlcInt i=0; i<n+1; i++) coeffs[i] = i;

m.add(IlcDistribute(vars, coeffs, vars));

// Contraintes redondantes
m.add(IlcScalProd(vars,coeffs) == n+1);
m.add(IlcSum(vars) == n+1);
```


4.3.2 Séquencement

Programme de licenciement chez Reneot : séquence de différentes configurations sur une chaîne de montage où chaque option ne peut être montée que sur un ratio fixé de véhicules.

		Configurations					
Option	Capacité	0	1	2	3	4	5
1	1/2	•				•	•
2	2/3			•	•		•
3	1/3	•				•	
4	2/5	•	•		•		
5	1/5			•			
	Demande	1	1	2	2	2	2

Les données :

```
IlcInt nbOptions = 5, nbConfs = 6, nbCars = 10;
IlcIntArray confs(m, 6, 0, 1, 2, 3, 4, 5);
IlcIntArray nbRequired(m, 6, 1, 1, 2, 2, 2, 2);
IlcIntArray maxSeq(m, 5, 1, 2, 1, 2, 1);
IlcIntArray overSeq(m, 5, 2, 3, 3, 5, 5);

IlcIntArray * optConf = new (m.getHeap()) IlcIntArray[5];
optConf[0] = IlcIntArray(m, 3, 0, 4, 5);
optConf[1] = IlcIntArray(m, 3, 2, 3, 5);
optConf[2] = IlcIntArray(m, 2, 0, 4);
optConf[3] = IlcIntArray(m, 3, 0, 1, 3);
optConf[4] = IlcIntArray(m, 1, 2);
```


Les variables :

```
IlcIntVarArray cars(m, nbCars, 0, nbConfs-1);
```

Contrainte de demande :

```
m.add (IlcDistribute(nbRequired, confs, cars));
```

MAIS, le premier argument de `IlcDistribute` doit être un tableau de variables :

```
IlcIntVarArray cards(m, 6);
for(IlcInt conf=0;conf<nbConfs;conf++){
  cards[conf]=IlcIntVar(m, nbRequired[conf],nbRequired[conf]);
}
m.add (IlcDistribute(cards, confs, cars));
```

Contraintes de capacité : pour chaque option, pour chaque sous-séquence il faut contraindre le nombre de véhicules nécessitant l'option.

Extraction de sous-séquence (documenté ?) :
`getImpl()->extract(IlcInt start, IlcInt length)`

Appartenance à un ensemble :

```
IlcConstraint IlcMember(const IlcIntExp exp,
                        const IlcIntArray elements);
```

MAIS *Solver Error* ! Il faut utiliser l'autre profil et utiliser une *variable ensembliste* :

```
IlcConstraint IlcMember(IlcIntExp element, IlcIntSetVar setVar);
```

Une variable ensembliste est fabriquée avec les valeurs *possibles*. On le peut le spécifier complètement en fixant son cardinal.


```
for (IlcInt opt=0; opt < nbOptions; opt++) {
 // Transformation du tableau en ensemble
  IlcIntSetVar set(m, optConf[opt]);
  m.add(IlcCard(set) == optConf[opt].getSize());

 // Pour toutes les sous-séquences
  for (IlcInt i=0; i < nbCars-overSeq[opt]+1; i++) {
    IlcIntVarArray carsseq = cars.getImpl()->extract(i,overSeq[opt]);

    IlcIndex j(m);
    m.add(IlcCard(j,IlcMember(carsseq[j], set)) <= maxSeq[opt]);
  }
}
```

4.4 IlcSequence

```
IlcConstraint IlcSequence(IlcInt nbMin,
                          IlcInt nbMax,
                          IlcInt seqWidth,
                          IlcIntVarArray vars,
                          IlcIntArray values,
                          IlcIntVarArray cards,
                          IlcFilterLevel level = IlcBasic);
```

Pour toutes les sous-séquences de `vars` de longueur `seqWidth`, il y a au moins `nbMin` et au plus `nbMax` valeurs parmi `values`. De plus, `cards[i]` est égal au nombre de valeurs de `vars` égales à `values[i]`.

Cette contrainte *ad hoc* permet de résoudre directement le problème de séquencement.

```
for (IlcInt opt=0; opt < nbOptions; opt++) {
  IlcInt s = optConf[opt].getSize();
  IlcIntVarArray cards(m, s);
  for(IlcInt i = 0; i < s; i++) {
    IlcInt nbconf = nbRequired[optConf[opt][i]];
     cards[i] = IlcIntVar(m, nbconf, nbconf);
  }
 m.add(IlcSequence(0, maxSeq[opt], overSeq[opt],
                   cars, optConf[opt], cards));
}
```

4.5 IlcPath

Calcul de chemin dans un graphe valué dont les nœuds sont indexés par des entiers.

```
IlcConstraint IlcPath(IlcIntVarArray next,
                      IlcFloatVarArray cumul,
                      IlcPathTransit transit,
                      IlcInt maxNbPaths,
                      IlcWhenEvent event = IlcWhenValue);
IlcPathTransit::IlcPathTransit(IlcPathTransitFunction func);
typedef IlcFloat (*IlcPathTransitFunction)(IlcInt, IlcInt);
```


On cherche p (= `maxNbPaths`) chemins dans un graphe à n sommets plus p sommets de départ plus p sommets d'arrivée.

`transit` est la fonction de valuation des arcs du graphe (poids).

`next` et `cumul` sont des tableaux à $n + 2 * p$ éléments :

- n sommets *généraux* ;
- p sommets d'arrivée ;
- p sommets de départ.

`next` est le tableau des *successeurs* : pour tout i, $next[i]$ est le successeur du nœud i sur un chemin (sauf pour $n \leq i < n + p$ où $next[i] = i + p$)

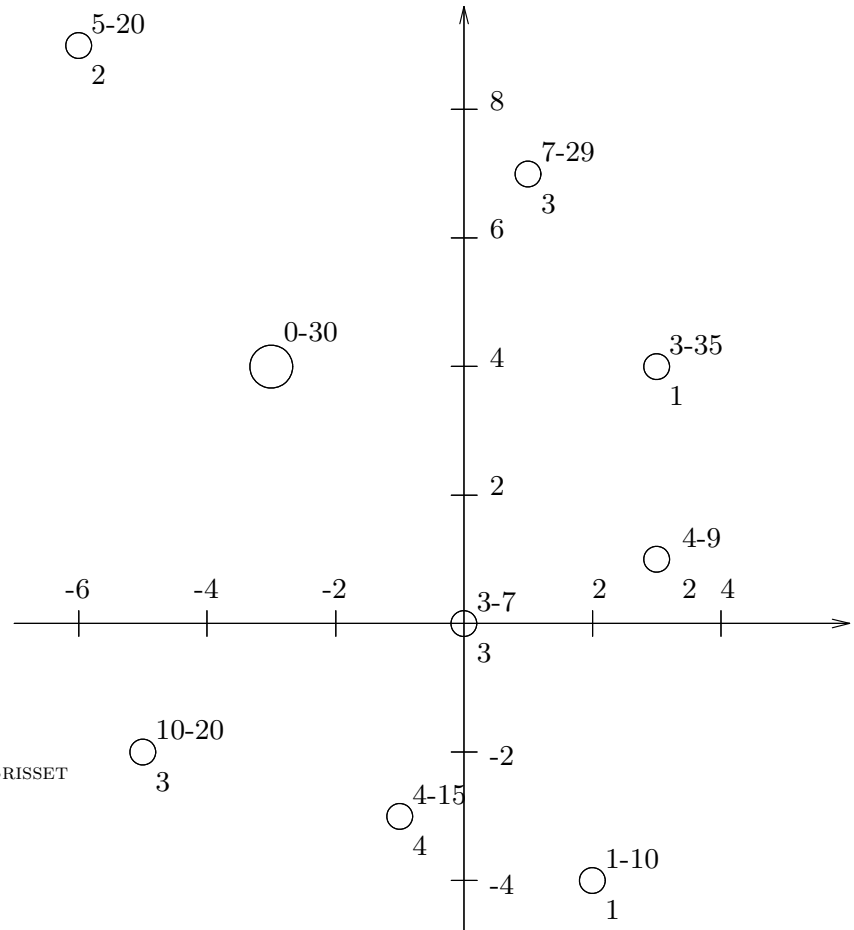
`cumul` est le tableau des cumuls des poids le long des chemins. Pour $0 \leq i < n$ ou $n + p \leq i < n + 2 * p$ on a $cumul[i] + transit[i, next[i]] \leq cumul[next[i]]$.

4.5.1 Tournées de véhicules

Soit

- un point de départ dans le plan (fournisseur) ;
- un ensemble de points (clients) positionnés dans le plan ((x, y)) ;
- des heures d'ouverture et de fermeture pour chaque client ;
- un ensemble de véhicules

Trouver des tournées pour les véhicules permettant de visiter chaque client en respectant les contraintes d'horaire. Minimiser la somme des longueurs des tournées.



Données :

```
const IlcInt NbNodes = 8;
const IlcInt NbPaths = 3;
const IlcInt N = NbNodes+2*NbPaths;
IlcIntArray     x(m, N, 0, 3, 2,-1,-5,-6, 1, 3,-3,-3,-3,-3,-3,-3);
IlcIntArray     y(m, N, 0, 1,-4,-3,-2, 9, 7, 4, 4, 4, 4, 4, 4, 4);
IlcIntArray first(m, N, 3, 4, 1, 4,10, 5, 7, 3, 0, 0, 0, 0, 0, 0);
IlcIntArray last (m, N, 7, 9,10,15,20,20,29,35,30,30,30,30,30,30);
IlcFloat GetDistance(IlcInt i, IlcInt j) {
  return IlcPower((x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])*(y[i]-y[j]),0.5);
}
```

Variables : chemins, dates et coût.

```
  IlcIntVarArray   next(m, N, 0, N-1);
  IlcFloatVarArray length(m, N, 0, last[NbNodes]); // Speed = 1
  for (i = 0 ; i < N ; i++) length[i].setRange(first[i], last[i]);
  IlcFloatVar total(m, 0, 60);
```


Contraintes : coût, dates et charges.

```
 // Coût des 3 routes
  m.add(length[NbNodes] + length[NbNodes+1] + length[NbNodes+2] == total);
  m.add(IlcPath(next,
                length,
                IlcPathTransit(m, GetDistance),
                NbPaths));
 // Contraintes redondantes
  for (i = 0 ; i < N ; i++)
    m.add(next[i] != i);
  m.add(IlcAllDiff(next));
```

Recherche et optimisation :

```
  m.add(IlcGenerate(next));
  m.add(IlcGenerate(length));
  m.setObjMin(total, 0.1);
```

4.5.2 Gestion de capacités

- chaque client a une certaine demande ;
- chaque véhicule a une certaine capacité.

Données :

```
const IlcFloat Capacity = 8;
IlcFloat demand[N] = { 3.,2.,1.,4.,3.,2.,3.,1.,0.,0.,0.,0.,0.,0.};
IlcFloat GetDemand(IlcInt i, IlcInt) { return demand[i]; }
```

Variables : charge véhicules

```
IlcFloatVarArray load(m, N, 0, Capacity);
```

Contraintes : capacité et demande satisfaites

```
m.add(IlcPath(next,
              load,
              IlcPathTransit(m, GetDemand),
              NbPaths));
```



```
sun20[45]% ./pathvrp.exe
ILOG Solver 4.310, licensed to "enac-toulouse"
The total cost is : 58.4
Route description :
-> 0 at 5.08 (0) -> 1 at 8.25 (3) -> 7 at 11.2 (5) -> Depot at 17.2 (6)
-> 2 at 9.43 (0) -> 3 at 12.6 (1) -> 4 at 16.7 (5) -> Depot at 23 (8)
-> 5 at 5.83 (0) -> 6 at 13.1 (2) -> Depot at 18.1 (5)
```

Effets des contraintes redondantes :

`Alldiff`	$\forall i\ next[i] \neq i$	Nb échecs	CPU
		7322	2.3
•		4478	1.73
	•	4499	1.33
•	•	2530	0.76

5 Méta-contrainte, réification

Toute contrainte peut être vue comme une fonction à valeur dans `IlcBool`.

Constructeur de *réification* de la classe `IlcBoolVar` :

```
class IlcBoolVar : public IlcConstraint {
  public:
    IlcBoolVar(const IlcConstraint exp);
};
```

À son tour, une variable booléenne peut être vue comme une expression entière :

```
class IlcIntExp {
  public:
    IlcIntExp(IlcConstraint bexp);
    IlcIntExp(IlcBoolVar bexp);
}
```


Exemple : pour un ensemble de tâches à effectuer et un ensemble de techniciens, choisir un technicien pour chaque tâche.

Données :

- d_i durée de la tâche i ;
- t_{ij} intervalle minimum de temps entre la fin de la tâche i et le début de la tâche j si elles sont effectuées par le même technicien.

Variables :

- X_i technicien effectuant la tâche i ;
- T_i date de début de la tâche i.

Contraintes sur la disponibilité des techniciens : pour toutes les paires de tâches i et j.

```
IlcBoolVar M, B1, B2;
m.add(M == (X[i] == X[j]));
m.add(T[i] + d[i] + t[i][j] <= T[j] + B1*IlcIntMax);
m.add(T[j] + d[j] + t[i][j] <= T[i] + B2*IlcIntMax);
m.add(M + B1 + B2 == 2);
```

M exprime que les tâches i et j sont effectuées par le même technicien ;

B1 permet de *relaxer* la précédence entre i et j ;

B2 permet de *relaxer* la précédence entre j et i.

Si les deux tâches ont des exécutants différents, les deux contraintes de précédences sont relaxées. Sinon, l'une d'elle doit être vérifiée (`B1=0` ou `B2=0`).

Ou encore avec une implication (inférieur) :

```
m.add((X[i] == X[j])
        <=
      ( T[i] + d[i] + t[i][j] <= T[j]
          ||
        T[j] + d[j] + t[i][j] <= T[i]));
```

6 Ensembles

6.1 Ensembles d'entiers

Codage par un vecteur de bits : min et max fixés.

```
class IlcIntSet{
  IlcIntSet(IlcManager m, IlcInt min, IlcInt max, IlcBool fullSet = IlcTrue);
  IlcIntSet(IlcManager m,const IlcIntArray values,IlcBool fullSet = IlcTrue);
  IlcBool isIn(IlcInt elt) const;
  IlcBool add(IlcInt elt);
  IlcBool remove(IlcInt elt);
};
```

Construction : initialisation du vecteurs de bits. Avec `fullset` à `IlcFalse`, on obtient l'ensemble vide.

Ajout : `add()` renvoie vrai si l'élément était possible et non déjà membre.

Retrait : `remove()` renvoie vrai si l'élément était membre.

La plupart des structure de donnés complexes d'ILOG Solver possède un *itérateur*.

Parcours des éléments d'un ensemble :

```
IlcInt val;
for(IlcIntSetIterator iter(set); iter.ok(); ++iter){
  val = *iter;
  // Utilisation de val
}
```

6.2 Variables

Une variable ensembliste prend ses valeurs dans un ensemble d'ensemble d'entiers.

Domaine d'une variable ensembliste :

- borne inférieure : ensemble des valeurs *nécessairement* dans l'ensemble ;
- borne supérieure : ensemble des valeurs *possiblement* dans l'ensemble.

La borne inférieure est incluse dans la borne supérieure.

Les deux bornes définissent un *treillis* de valeurs possibles pour la variable.

Variable liée : nécessaires = possibles.


```
class IlcIntSetVar {
  IlcIntSetVar(IlcManager m, IlcInt min, IlcInt max);
  IlcIntSetVar(IlcManager m, const IlcIntArray array);
  IlcIntSet getPossibleSet() const;
  IlcIntSet getRequiredSet() const;
  void addRequired(IlcInt elt) const;
  void removePossible(IlcInt elt) const;
  ...
};
```

L'ensemble initial des valeurs nécessaires est vide.

6.3 Expressions

```
IlcIntSetVar IlcIntersection(IlcIntSetVar var1, IlcIntSetVar var2);
IlcIntVar IlcCard(IlcIntSetVar var);
IlcIntSetVar IlcSetOf(IlcIndex& i, IlcConstraint ct);
```

Exemple :

```
IlcIntSetVarArray X; IlcIndex i;
IlcIntSetVar s = IlcSetOf(i, IlcCard(X[i]) == 2);
```

6.4 Contraintes

```
IlcConstraint IlcNullIntersect(IlcIntSetVar a, IlcIntSetVar b);
IlcConstraint IlcAllNullIntersect(IlcIntSetVarArray array);
IlcConstraint IlcSubset(IlcIntSetVar a, IlcIntSetVar   b);
IlcConstraint IlcSubsetEq(IlcIntSetVar a, IlcIntSetVar b);
IlcConstraint IlcMember(IlcIntExp element, IlcIntSetVar setVar);
```


6.5 Bin Packing

Répartition dans des boîtes de même taille.

Somme des éléments d'un ensemble (prédéfinie dans Solver 4.4) :

```
IlcIntVar SetSum(IlcIntSetVar Set, IlcIntArray Vals) {
  IlcIntVarArray Members(Set.getManager(), Vals.getSize());
  for(IlcInt j = 0; j < Vals.getSize(); j++) {
      Members[j] = (IlcMember(Vals[j], Set));
  }
  IlcIntVar Sum = IlcScalProd(Members, Vals);
  return Sum;
}
```

Variables :

```
IlcInt Size = 5;
IlcIntArray Vals(m, Size, 1, 2, 3, 4, 5);
IlcIntSetVar Set(m,  Vals);
IlcIntSetVarArray Parts(m,  Size, Set);
IlcIntVar Sum(m, 1, IlcSum(Vals));
```

Contraintes : partition et sommes identiques.

```
m.add(IlcAllNullIntersect(Parts));
IlcIntSetVar unnion = Parts[0];
for(IlcInt j = 1; j < Size; j++)
  unnion = IlcUnion(unnion, Parts[j]);
m.add(unnion == IlcIntSet(Vals));
for(IlcInt i = 0; i < Size; i++){
  IlcIntVar EachSum = SetSum(Parts[i], Vals);
  m.add(EachSum == Sum || EachSum == 0);
};
```


Ordonnancement des parties et recherche de solution.

```
for(i = 1; i < Size; i++) {
  m.add(IlcCard(Parts[i]) <= IlcCard(Parts[i-1]));
}
m.add(IlcInstantiate(Sum));
m.add(IlcGenerate(Parts));
```

7 Variables réelles

Il n'est pas possible de calculer une solution exacte pour un système d'équations en nombres réels.

À une variable réelle est associé un intervalle $[min, max]$.

Une variable réelle est *liée* pour une précision donnée si :

$$\frac{max - min}{max(1, |min|)} \leq precision$$

```
IlcManager::setDefaultPrecision(IlcFloat precision);
```

La fonction

```
void IlcInitFloat();
```

doit être appelée avant toute utilisation des nombres réels avec Solver.


```
class IlcFloatVar : public IlcFloatExp {
  IlcFloatVar(IlcManager m,
              IlcFloat min,
              IlcFloat max,
              const char* name = 0);
  IlcFloatVar(IlcManager m,
              IlcFloat min,
              IlcFloat max,
              IlcFloat precision,
              const char* name = 0);
  IlcFloatVar(IlcIntVar var);
  IlcFloat getMin() const;
  IlcFloat getMax() const;
...
};
```

7.1 Expressions et contraintes

Arithmétiques comme en entier avec en plus

```
IlcFloatExp IlcLog(const IlcFloatExp x);
IlcFloatExp IlcPower(const IlcFloatExp  x, const IlcFloat p);
IlcFloatExp IlcPower(const IlcFloatExp  x, const IlcInt p);
IlcFloatExp IlcSin(const IlcFloatExp x);

IlcConstraint IlcNull(const IlcFloatExp x);
...
```


7.2 Recherche de solution, optimisation

Par dichotomie

```
IlcGoal IlcInstantiate(const IlcFloatVar var,
                       IlcBool increaseMinFirst=IlcTrue,
                       IlcFloat prec=0);
IlcGoal
  IlcGenerate
    (const IlcFloatVarArray,
     IlcChooseFloatIndex chooseVariable=IlcChooseFirstUnboundFloat,
     IlcBool increaseMinFirst=IlcTrue,
     IlcFloat prec=0);
void IlcManager::setObjMin(IlcFloatVar obj, IlcFloat step);
```

7.3 Numerica

Numerica permet de gérer les équations non linéaires (Solver attend la linéarisation avant de les traiter).

Résolution de

$$\begin{aligned} x^2 + y^2 &= 1 \\ x^2 &= y \end{aligned}$$

Numerica est un objet à qui on attache les contraintes non linéaires.

```
IlcFloatVar x(m, -10, 10);
IlcFloatVar y(m, -10, 10);
IlcNumerica num(m);             // create an instance  of IlcNumerica
// add the two constraints to the instance of IlcNumerica
num.add(IlcSquare(x) + IlcSquare(y) == 1);
num.add(IlcSquare(x) == y);
num.close();                    // close the system
```



```
IlcFloatVarArray vars(m,2,x,y);
m.add(IlcBestGenerate(vars, IlcChooseMaxSizeFloat));

while (m.nextSolution()) {
  m.out() << "x: " << x << endl;
  m.out() << "y: " << y << endl;
}
```

Deux solutions trouvées :

```
x: [0.786151377740..0.786151377791]
y: [0.618033988724..0.618033988796]

x: [-0.786151377791..-0.786151377740]
y: [0.618033988724..0.618033988796]
```

8 Contraintes utilisateur

De nouvelles contraintes peuvent être définies par l'utilisateur :

- propagation de l'information d'une variable à l'autre ;
- conditions de réveil.

8.1 Accès et modifications du domaine d'une variable

Les modifications sont correctes uniquement sur des variables : le domaine d'une expression n'est pas représenté.


```
class IlcIntExp {
    IlcInt getMax() const;
    IlcInt getMin() const;
    IlcInt getNextHigher (IlcInt val) const;

    IlcInt getSize () const;
    IlcBool isBound() const;
    IlcBool isInDomain (IlcInt val) const;

    void removeRange(IlcInt min, IlcInt max) const;
    void removeValue(IlcInt value) const;
    void setMax(IlcInt max) const;
    void setMin(IlcInt min) const;
    void setRange (IlcInt min, IlcInt max) const;
    void setValue(IlcInt value) const;
};
```

8.2 Nouvelle classe de contrainte

Héritage de la classe `ConstraintI` :

```
class MyConstraint : public IlcConstraintI {
     ... // paramètres de la constrainte
public :
    MyConstraint(IlcManager m, ... args ...);
    void post();
    void propagate();
    IlcBool isViolated() const;
}
```

post Action lorsque la contrainte est posée (`m.add()`) ;

propagate Action lorsque la contrainte est réveillée.

Exemple : nombres premiers entre eux.

```
class AreNotMultipleI : public IlcConstraintI {
private:
  IlcIntExp _x;
  IlcIntExp _y;
public:
  AreNotMultipleI(IlcManager m, IlcIntExp x, IlcIntExp y);
  void post();
  void propagate();
};

IlcConstraint AreNotMultiple(IlcIntExp x, IlcIntExp y) {
  IlcManager m = x.getManager();
  return new (m.getHeap()) AreNotMultipleI(m.getImpl(), x, y);
}
```

Constructeur : utilisation des constructeurs de la classe héritée et des attributs.

```
AreNotMultipleI::AreNotMultipleI(IlcManager m,
                                 IlcIntExp x,
                                 IlcIntExp y):
  IlcConstraintI(m), _x(x), _y(y){}
```

Action lorsque la contrainte est posée : choix des conditions de réveil.

```
void AreNotMultipleI::post() {
  _x.whenValue(this);
  _y.whenValue(this);
}
```

Trois conditions de réveil possibles :

```
void IlcIntExp::whenDomain(const IlcGoal ct) const;
void IlcIntExp::whenRange(const IlcGoal ct) const;
void IlcIntExp::whenValue(const IlcGoal ct) const;
```


Propagation : prise en compte de la symétrie

```
void propagateAux(IlcIntExp x, IlcIntExp y) {
  IlcInt xval = x.getValue();
  for (IlcIntExpIterator iter(y); iter.ok(); ++iter)
    if ((*iter) % xval == 0)
      y.removeValue(*iter);
}

void AreNotMultipleI::propagate(){
  if (_x.isBound())
    propagateAux(_x, _y);
  else                    // _y est liée à cause des conditions de réveil
    propagateAux(_y, _x);
}
```

Un échec sera déclenché (`IlcManager::fail()`) si un domaine est réduit à vide.

Raffinement pour éviter le test dans la méthode `propagate` : utilisation d'un *démon*, i.e. un but (`IlcGoal`) exécuté *immédiatement*.

On peut fabriquer un démon avec les macros `ILCDEMON` :

```
ILCDEMON2(Demon, IlcIntExp, x, IlcIntExp, y) {
  propagateAux(x, y);
}
```

L'action lors de la pause de la contrainte devient alors :

```
void AreNotMultipleI::post() {
  _x.whenValue(Demon(getManager(), _x, _y));
  _y.whenValue(Demon(getManager(), _y, _x));
}
```


8.3 Métacontraintes

Pour que la contrainte puisse être réifiée, d'autres méthodes doivent être définies :

- `makeOpposite()` doit retourner une implémentation de la négation de la contrainte (`AreMultiple(_x,_y)`) ;
- `metaPost(IlcConstraint meta)` doit associer `meta` aux variables de la contrainte pour les mêmes conditions de réveil que `post` ;
- `isViolated()` doit renvoyer `IlcTrue` si la contrainte ne peux pas être satisfaite (pas nécessairement `IlcFalse` dans le cas contraire).

9 Construction de but

Le *manager* gère une pile de buts qui permet une gestion non-déterministe.
Un but peut *réussir* ou *échouer* (appel à `IlcManager::fail()`).

9.1 Définition d'un but avec les macros ILCGOAL

Affichage d'une variable entière :

```
ILCGOAL1(Print, IlcIntVar, x) {
  getManager().out() << x << endl;
  return 0;
}
```


9.2 Définition d'un but *à la main*

Héritage de la classe `IlcGoalI`, définition du fonctionnement (`execute()`)
et instanciation pour obtenir une fonction.

```
class PrintI : public IlcGoalI {
  IlcIntVar _x;
public:
  PrintI(IlcManagerI* m, IlcIntVar x);
  IlcGoal execute();
}
PrintI::PrintI(IlcManagerI* m, IlcIntVar x): IlcGoalI(m), _x(x) {}
IlcGoal PrintI:execute() {
  getManager().out() << x << endl;
  return 0;
}
IlcGoal Print(IlcManager m, IlcIntVar x) {
  return new (m.getHeap()) PrintI(m.getImpl(), x); }
```

9.3 Sous-buts

La valeur retournée par la méthode `execute()` est un but.

`IlcAnd` permet de composer les buts pour une exécution en séquence.

```
ILCGOAL2(Instantiate2, IlcIntVar, x, IlcIntVar, y) {
  return(IlcAnd(IlcInstantiate(x), IlcInstantiate(y)));
}
```

`IlcOr` permet de composer les buts pour une exécution non-déterministe avec création de point de choix.

9.4 But récursif

Instanciation non déterministe d'une variable :

```
ILCGOAL1(IlcIntInstantiate, IlcIntVar, var) {
  if (var.isBound()) return 0;
  IlcInt val = var.getMin();
  return IlcOr(var == val, IlcAnd(var != val, this));
}
```

Instanciation d'un tableau de variables :

```
ILCGOAL2(IlcIntGenerate,
         IlcIntVarArray, vars,
         IlcChooseIntIndex, chooseIndex) {
  IlcInt index = chooseIndex(vars);
  if(index == -1) return 0;
  return IlcAnd(IlcIntInstantiate(vars[index], select), this);
}
```

10 Scheduler

ILOG Scheduler est une librairie utilisant Solver offrant des objets et des contraintes spécifiques aux problèmes d'ordonnancements et d'allocation de ressources.

Nouvelles classes :

- `IlcSchedule` : ordonnancement de tâches utilisant des ressources ;
- `IlcActivity` : tâche d'une durée donnée, ordonnées, utilisant des ressources ;
- `IlcRessource` : ressource ;
- `IlcAltResSet` : ensemble de ressources.

Problèmes concernés :

Ordonnancement pur : contraintes temporelles uniquement entre des tâches dont l'ensemble des ressources nécessaires est connu.

Allocation pure : choix des ressources pour des tâches fixées dans le temps.

Problème général : choix des ressources et des dates d'exécution.

10.1 Activité

`IlcActivity`

↪ `IlcIntervalActivity` Activité continue pendant une période de temps, éventuellement de longueur inconnue.

↪ `IlcBreakableActivity` Activité interruptible par des pauses.

10.2 Contrainte temporelles

```
class IlcActivity  {
  IlcTimeBoundConstraint startsAfter(IlcInt time);
  IlcPrecedenceConstraint startsAfterEnd(IlcActivity act, IlcInt delay = 0);
  IlcPrecedenceConstraint startsAfterStart(IlcActivity act, IlcInt delay = 0);
  IlcTimeBoundConstraint startsAt(IlcInt time);
  IlcPrecedenceConstraint startsAtEnd(IlcActivity act, IlcInt delay = 0);
  ... }
```


10.3 Ressource

`IlcResource`

↪ `IlcCapResource` : ressource capacitive

↪ `IlcDiscreteResource` : capacité entière qui peut varier avec le temps. À chaque instant, la somme des demandes sur cette ressource ne peux dépasser la capacité.

↪ `IlcUnaryResource` : capacité unitaire. À chaque instant, la ressource ne peut être utilisée que par une activité. Une ressource unitaire peut être agrémentée de *temps de transition* minimum entre deux utilisations.

↪ `IlcDiscreteEnergy` : produit de la capacité utilisée par le temps d'utilisation. La limite est posée par intervalle de temps.

↪ `IlcReservoir` : ressource intermédiaire. La ressource peut être consommée et alimentée par les activités. Il est assuré que la ressource est plus alimentée que consommée (et éventuellement pas trop alimentée).

↪ `IlcStateResource` : ressource à état. De capacité infinie, elle assure que deux ressources nécessitant des états différents ne peuvent pas s'exécuter en même temps.

10.4 Relation entre activité ressource

Une activité :

nécessite une ressource (`IlcUnaryResource`) ;

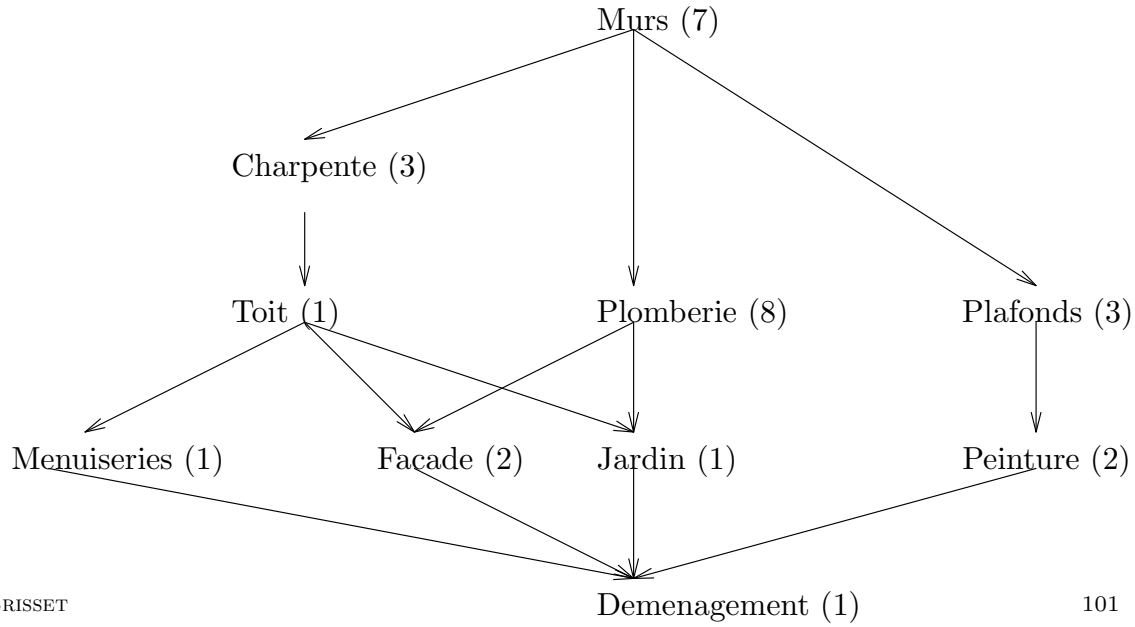
consomme une ressource (`IlcCapResource`) ;

alimente une ressource (`IlcReservoir`).

```
class IlcIntervalActivity : public IlcActivity {
IlcResourceConstraint
    requiresState(IlcStateResource resource,
                  IlcAny state,
                  IlcTimeExtent extent = IlcFromStartToEnd,
                  IlcBool outward = IlcTrue);
IlcResourceConstraint consumes(IlcCapResource resource, IlcInt capacity = 1);
IlcResourceConstraint produces(IlcReservoir reservoir, IlcInt capacity = 1);
}
```

11 Construire une maison

11.1 Ordonnancement simple




```
#include <ilsched/ilsched.h>
void after(IlcIntervalActivity a, IlcIntervalActivity b) {
  a.getManager().add(a.startsAfterEnd(b));
}
typedef IlcIntervalActivity IIA;
main() {
 // Initialisation
  IlcManager m(IlcEdit);
  IlcInt horizon = 29;
  IlcSchedule schedule(m, 0, horizon);

 // Activités
  IIA masonry(schedule, 7); masonry.setName("masonry");
  IIA carpentry(schedule, 3); carpentry.setName("carpentry");
  IIA roofing(schedule, 1); roofing.setName("roofing");
  IIA plumbing(schedule, 8); plumbing.setName("plumbing");
  ...
```

```
 // Contraintes
  after(capentry, masonry); after(plumbing, masonry);
  after(ceiling, masonry); after(roofing, capentry);
  ...
 // Critère d'optimisation, recherche
  IlcIntVar makespan = moving.getendVariable();
  m.add(IlcInstantiate(makespan));
  m.setObjMin(makespan);

  m.nextSolution();

 // Affichage
  for(IlcActivityIterator iter(schedule); iter.ok(); ++iter) {
    m.out() << *iter << endl;
  }
}
```



```
makespan = [18]
moving    [17 -- 1 --> 18]
garden    [15..16 -- 1 --> 16..17]
facade    [15 -- 2 --> 17]
windows   [11..16 -- 1 --> 12..17]
painting  [10..15 -- 2 --> 12..17]
roofing   [10..14 -- 1 --> 11..15]
ceiling   [7..12 -- 3 --> 10..15]
plumbing  [7 -- 8 --> 15]
carpentry [7..11 -- 3 --> 10..14]
masonry   [0 -- 7 --> 7]
```

11.2 Coût

- La journée de travaile coûte 1000 ;
- il faut payer au jour le jour ;
- on ne dispose de pas plus de 2000 par jour.

Utilisation d'une ressource capacitive :

```
IlcDiscreteResource budget(schedule, 2000);

// Contraintes sur chaque activité
for(IlcIntervalActivityIterator ite(schedule); ite.ok(); ++ite) {
  IlcIntervalActivity activity = *ite;
  m.add(activity.requires(budget, 1000));
}
```


Étiquettage : choix des départs de chaque tâche.

```
IlcIntVarArray starts(m, schedule.getNumberOfActivities());
int n = 0;
for(IlcIntervalActivityIterator ite1(schedule); ite1.ok(); ++ite1, ++n) {
  starts[n] = (*ite1).getStartVariable();
}
m.add(IlcGenerate(starts));
```

Solution optimale trouvée immédiatement :

```
makespan = [19]
moving[18 -- 1 --> 19]
windows[11 -- 1 --> 12]
facade[15 -- 2 --> 17]
painting[15 -- 2 --> 17]
garden[17 -- 1 --> 18]
ceiling[12 -- 3 --> 15]
...
```

Pas plus de 10000 par semaine : utilisation d'une contrainte énergétique.

```
IlcDiscreteEnergy budget_semaine(schedule, 7, 10000);
for(IlcIntervalActivityIterator ite2(schedule); ite2.ok(); ++ite2) {
  m.add((*ite2).consumes(budget_semaine, 1000));
}
```

```
makespan = [23]
moving[22 -- 1 --> 23]
windows[11 -- 1 --> 12]
facade[17 -- 2 --> 19]
painting[17 -- 2 --> 19]
garden[21 -- 1 --> 22]
ceiling[14 -- 3 --> 17]
plumbing[9 -- 8 --> 17]
roofing[10 -- 1 --> 11]
carpentry[7 -- 3 --> 10]
masonry[0 -- 7 --> 7]
```


11.3 Ouvriers

Deux ouvriers participent à la construction. Ils constituent des ressources *unaires*, alternatives, parmi lesquelles chaque tâche doit choisir.

Deux ressources unitaires `IlcUnaryResource` et un ensemble de ressources `IlcAltResSet` :

```
IlcUnaryResource Joe(schedule); Joe.setName("Joe");
IlcUnaryResource Jim(schedule); Jim.setName("Jim");
IlcAltResSet workers(schedule, 2, Jim, Joe);
// Désignation des contraintes d'utilisation
IlcAltResConstraint* cs =
  new (m.getHeap()) IlcAltResConstraint [schedule.getNumberOfActivities()];
n = 0;
for(IlcIntervalActivityIterator ite(schedule); ite.ok(); ++ite, ++n) {
  cs[n] = (*ite).requires(workers, 1);
  m.add(cs[n]);
}
```

À chaque choix de ressource dans l'ensembles des alternatives est associée une variable ... qu'il faut étiquetter :

```
m.add(IlcAssignAlternative(workers));

while(m.nextSolution()) {
  m.out() << "makespan = " << makespan << endl;
  n = 0;
  for(IlcActivityIterator iter(schedule); iter.ok(); ++iter,++n) {
    m.out() << *iter << " par " << cs[n].getSelected().getName() << endl;
  }
}
```



```
makespan = [29]
moving[28 -- 1 --> 29] par Jim
windows[11 -- 1 --> 12] par Jim
facade[20 -- 2 --> 22] par Jim
painting[25 -- 2 --> 27] par Jim
garden[27 -- 1 --> 28] par Jim
ceiling[22 -- 3 --> 25] par Jim
plumbing[12 -- 8 --> 20] par Jim
roofing[10 -- 1 --> 11] par Jim
...
makespan = [19]
moving[18 -- 1 --> 19] par Jim
windows[11 -- 1 --> 12] par Jim
facade[15 -- 2 --> 17] par Joe
painting[15 -- 2 --> 17] par Jim
garden[17 -- 1 --> 18] par Jim
...
```

11.4 Entreprise

Une entreprise fournit des ouvriers qui travaillent pendant des périodes continues. À tout instant, un nombre maximal d'ouvriers sur le chantier est fixé.

```
int nb_workers = 5, max_nb_workers = 3;
IlcReservoir group(schedule, max_nb_workers);
IlcIntervalActivity *workers =
  new (m.getHeap()) IlcIntervalActivity [nb_workers];
for(IlcInt w = 0; w < nb_workers; w++) {
  workers[w] = IlcIntervalActivity(schedule);
  m.add(workers[w].provides(group, 1));
}
for(w = 0; w < nb_workers; w++) {
  m.add(IlcInstantiate(workers[w].getDurationVariable()));
  m.add(IlcInstantiate(workers[w].getStartVariable()));
}
```


12 Classes et méthodes de ILOG Scheduler

12.1 Objet primal

```
class IlcSchedule {
  IlcSchedule(IlcManager m, IlcInt timeMin, IlcInt timeMax);
  IlcInt getNumberOfActivities() const;
  IlcInt getNumberOfResources() const;
  ...
};
```

12.2 Activité

```
class IlcActivity  {
  IlcInt getDurationMax() const;
  IlcIntVar getDurationVariable() const;
  IlcIntVar getStartVariable() const;
  IlcIntVar getProcessingTimeVariable() const;
  ...
  void setDuration(IlcInt duration);
  ...
```

Le `Duration` peut différer du `ProcessingTime` pour les tâches interruptibles.

Constructeurs :

```
class IlcIntervalActivity : public IlcActivity {
 public:
  IlcIntervalActivity(const IlcSchedule schedule);
  IlcIntervalActivity(const IlcSchedule schedule, IlcInt duration);
  IlcIntervalActivity(const IlcSchedule schedule, IlcIntVar durationVariable);
  IlcIntervalActivity(const IlcSchedule schedule,
                      IlcIntVar startVariable,
                      IlcIntVar endVariable,
                      IlcIntVar durationVariable);
```

NB : une activité est attachée à un *schedule* et non pas au *manager*.

Utilisation d'une ressource :

```
IlcResourceConstraint consumes(IlcCapResource resource,
                               IlcInt capacity = 1);
IlcResourceConstraint
  requires(IlcCapResource resource,
           IlcInt capacity = 1,
           IlcTimeExtent extent = IlcFromStartToEnd,
           IlcBool outward = IlcTrue);
```

- `consumes` exprime une consommation au début de l'activité : quantité.
- `requires` exprime un besoin instantané : puissance.

Alimentation d'un `IlcReservoir` :

```
IlcResourceConstraint produces(IlcReservoir reservoir,
                               IlcInt capacity = 1);
IlcResourceConstraint
  provides(IlcReservoir reservoir,
           IlcInt capacity = 1,
           IlcTimeExtent extent = IlcFromStartToEnd,
           IlcBool outward = IlcFalse);
```

- `produces` produit une quantité consommable rendue disponible à la fin de la tâche.
- `provides` alimente par unité de temps une capcité donnée (puissance).

12.3 Ressource

Capacité éventuellement variable au cours du temps.

```
class IlcDiscreteResource : public IlcCapResource {
  IlcDiscreteResource(IlcSchedule schedule,
                      IlcInt capacity,
                      IlcBool timetable = IlcTrue);
  void setCapacityMax(IlcInt timeMin,
                      IlcInt timeMax,
                      IlcInt capacityMax);
  void setCapacityMin(IlcInt timeMin,
                      IlcInt timeMax,
                      IlcInt capacityMin);
  IlcInt getCapacityMax(IlcInt time) const;
  IlcInt getCapacityMaxMin(IlcInt timeMin, IlcInt timeMax) const;
  ...
};
```


Capacité par intervalle de temps.

```
class IlcDiscreteEnergy : public IlcCapResource {
  IlcDiscreteEnergy(IlcSchedule schedule,
                    IlcInt timeStep,
                    IlcInt energy,
                    IlcBool timetable = IlcTrue);
  IlcInt getEnergy() const;
  IlcInt getEnergyMax(IlcInt time) const;
  void setEnergyMin(IlcInt timeMin,
                    IlcInt timeMax,
                    IlcInt energyMin);
...
};
```

NB : le temps total du *schedule* doit être un multiple de `timeStep`.

Exemple : pas plus de 5 jour de travail par semaine, au moins 8h de sommeil par nuit, ...

Ressource consommable et alimentable : stock entre deux activités.

```
class IlcReservoir : public IlcCapResource {
  IlcReservoir(IlcSchedule schedule,
               IlcInt capacity = IlcIntMax/2,
               IlcInt initialLevel = 0,
               IlcBool timetable = IlcTrue);

  void setCapacityMax(IlcInt timeMin,
                      IlcInt timeMax,
                      IlcInt capacityMax);
}
```

NB : la capacité maximale n'est jamais dépassée.

La méthode `IlcResource::close` doit impérativement être appelée pour que les contraintes sur un réservoir soient effectives.

Ensemble de ressources alternatives :

```
class IlcAltResSet{
  IlcAltResSet(IlcSchedule schedule, IlcInt size);
  IlcAltResSet(IlcSchedule schedule, IlcInt size,
               const IlcResource resource0,
               const IlcResource resource1 ...);
  void makeRedundantResource(IlcBool timetable = IlcFalse);
  IlcCapResource getRedundantResource() const;
};
```

Ressource redondante : somme des ressources de l'ensemble de ressources.

13 Arc-consistance

Définition 1 (Support d'une valeur) *Soit un CSP binaire* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *et* $c = (\{x, y\}, \mathcal{R}_c) \in \mathcal{C}$ *une contrainte de* $\mathcal{P}$ *sur les variables* x *et* y*. Une valeur* $v_y \in \mathcal{D}_y$ *du domaine de* y *est un* support *pour une valeur* $v_x \in \mathcal{D}_x$ *du domaine de* x *si et seulement si* $(v_x, v_y) \in \mathcal{R}_c$*, i.e.* (v_x, v_y) *satisfait* c*.*

Définition 2 (Arc-consistance) *Un arc orienté* (x, y) *du graphe associé à un CSP* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$*, et correspondant à une contrainte* $c = (\{x, y\}, \mathcal{R}_c) \in \mathcal{C}$*, est* arc-consistant *si et seulement si toutes les valeurs du domaine de* x *ont un support dans le domaine de* y*.*

Définition 3 (Arc-consistance d'un CSP binaire) *Un CSP binaire* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *est* arc-consistant *si et seulement si* $\forall c = (\{\{x, y\}, \mathcal{R}_c) \in \mathcal{C}$*, les arcs* (x, y) *et* (y, x) *sont arc-consistants.*

Procédure d'établissement de l'arc-consistance d'une variable.

révise(x, y)

1: $modif := faux$
2: **for all** $v_x \in \mathcal{D}_x$ **do**
3: **if** $\nexists v_y \in \mathcal{D}_y$ telle que $(v_x, v_y) \in \mathcal{R}_{(x,y)}$ **then**
4: $\mathcal{D}_x := \mathcal{D}_x / \{v_x\}$
5: $modif := vrai$
6: **end if**
7: **end for**
8: return $modif$

Procédure d'établissement de l'arc-consistance d'un CSP binaire.

AC3

1: $\mathcal{Q} := \bigcup_{c=(\{x,y\},\mathcal{R}_c)\in\mathcal{C}}\{(x,y),(y,x)\}$
2: **while** $\mathcal{Q} \neq \emptyset$ **do**
3: $(x,y) \in \mathcal{Q}$
4: $\mathcal{Q} := \mathcal{Q}/(x,y)$
5: **if** révise(x,y) **then**
6: $\mathcal{Q} := \mathcal{Q} \cup \{(z,x),\ \exists c = (\{x,z\},\mathcal{R}_c) \in \mathcal{C},\ z \neq y\}$
7: **end if**
8: **end while**

Définition 4 (Arc-consistance généralisée) *Soit un CSP* $\mathcal{P} = (\mathcal{X},\mathcal{D},\mathcal{C})$ *et une contrainte* $c = (\mathcal{X}_c,\mathcal{R}_c) \in \mathcal{C}$. *On dit que :*

- *un tuple* $\tau \in \mathcal{R}_c$ *est* valide *si et seulement si* $\forall x \in \mathcal{X}_c,\ \pi_x(\tau) \in \mathcal{D}_x$;
- *une valeur* $v_x \in \mathcal{D}_x$ *d'une variable* $x \in \mathcal{X}_c$ *est* consistante avec c *si et seulement si* $\exists\tau \in \mathcal{R}_c$ *tel que* $\pi_x(\tau) = v_x$ *et que* τ *soit valide ;*
- *la contrainte* c *est* arc-consistante *si et seulement si* $\forall x \in \mathcal{X}_c,\ \mathcal{D}_x \neq \emptyset$ *et* $\forall v_x \in \mathcal{D}_x,\ v_x$ *est consistante avec* c.

Le problème $\mathcal{P}$ *est* arc-consistant *si et seulement si toutes ses contraintes sont arc-consistantes.*

Définition 5 (k-consistance) *Un CSP* $\mathcal{P} = (\mathcal{X},\mathcal{D},\mathcal{C})$ *est* k-consistant *si et seulement si* $\forall\mathcal{I} = (\mathcal{X}_\mathcal{I} = \{x_1,\ldots,x_{k-1}\},\mathcal{V}_\mathcal{I})$, *une instanciation partielle consistante de* $k-1$ *variables, et* $\forall x \in \mathcal{X}/\mathcal{X}_\mathcal{I}$, *une variable qui n'est pas instanciée par* $\mathcal{I}$, *alors* $\exists v \in \mathcal{D}_x$ *telle que* $\mathcal{I} \cup \{(x,v)\}$ *soit consistante.*

Un CSP est fortement k-consistant *si et seulement s'il est* i-*consistant* $\forall i \in [1,k]$.