

CPLEX

Introduction

Avant de commencer:

ILOG CONCERT	<i>Solveurs</i>	
	CPLEX	programmation mathématique (LP, QP, MILP, MIQP)
	CP	programmation par contraintes

- Une façon commune de définir le modèle (environnement CONCERT)
- Toutes les classes sont liées à l'environnement
- Les composantes (variables/contraintes/objectif) sont ajoutées au modèle
- Le modèle est "extrait" par CPLEX

Classes de base pour résoudre un modèle:

	Étapes	Classes
CONCERT	environnement (classe centrale)	IloEnv
	modèle	IloModel
	variables	IloNumVar, IloNumColumn, IloNumVarArray, IloConversion
	contraintes	IloRange, IloExpr, IloRangeArray
	objectif	IloObjective
CPLEX	résolution	IloCplex

- Toutes les classes (sauf IloCplex) sont répertoriées dans la section Concert du guide de référence de la documentation en ligne
- La classe IloCplex comporte aussi une section "avancée" qui permet d'intervenir pendant la résolution du branch&cut ==> callbacks

IMPORTANT:

- Par défaut, CPLEX (tout comme Gurobi, Matlab, etc.) va utiliser **TOUTES** les ressources (coeurs/CPU) de la machine sur laquelle le programme tourne.
- **Pour limiter le valeur de threads utilisés par CPLEX, on vous demande d'utiliser l'instruction suivante qui en fixe la limite à 1:**

```
cplex.setParam(IloCplex::Param::Threads, 1);
```

Interfaces:

Interactive	charger un modèle sur disque et le résoudre en variant les paramètres manuellement
Librairie	construire et résoudre un modèle avec l'aide de classes et fonctions en divers langages (C, C++, Java, Python, .NET, MATLAB)
OPL	langage de modélisation adapté (inclut interaction avec Excel)

Compilation:

	Fichiers d'entête	Directives au préprocesseur (-D)
C	#include <ilcplex/cplex.h>	
C++	#include <ilcplex/ilocplex.h>	IL_STD ==> accès à la STL NDEBUG ==> assert (vérifications) inactives
Java	import ilog.concert.*; import ilog.cplex.*;	

	Librairies
C	libcplex.a libpthread.a libm.a
C++	libilocplex.a libconcert.a libcplex.a libpthread.a libm.a
Java	cplex.jar

N.B.: En C++, l'ordre des librairies en caractères gras est important!!!

Macro (C++): **ILOSTLBEGIN** (==> *using namespace std;*)

- À utiliser avant tout appel aux classes de CPLEX.
- Pourrait être étendue dans l'avenir pour inclure d'autres fonctionnalités.

CPLEX en C++ (Concert):

Tour d'horizon :

- Classes, définitions et types de base:

IloExtractable	classe mère pour les classes de construction du modèle
IloAlgorithm	classe mère pour les classes de résolution de modèle
IloNum, IloInt IloBool	alias pour double, int, bool (représenté comme int)
ILOFLOAT ILOINT ILOBOOL	types des variables continues, entières, binaires
IloInfinity	infini (bornes pour variables/contraintes)

- Environnement et modèle

IloEnv	créer un environnement commun (création + résolution du modèle)
IloModel	modéliser LP, QP, MILP, MIQP (contient variables, contraintes, etc.)

- Objectif:

IloObjective	définir la fonction objectif
IloMinimize, IloMaximize	définir la direction d'optimisation

- Variables:

IloNumColumn	définir des variables en ajoutant coefficients un à un
IloNumVar IloIntVar IloBoolVar	déclarer variables continues (défaut), entières, binaires
IloConversion	convertir les variables continues en d'autres types

- Contraintes:

IloExpr	construire une expression en ajoutant coefficients un à un
IloRange	déclarer des contraintes
IloSum	faire la somme d'un tableau de variables
IloScalProd	faire le produit scalaire d'un tableau de variables et d'un tableau de coefficients
IloIfThen	modéliser des contraintes "Big M" ou des contraintes logiques

- Tableaux + macro:

IloArray	Patron de classe pour créer des tableaux extensibles multi-dimensions
IloExtractableArray	tableau extensible de IloExtractable
IloNumArray IloIntArray IloBoolArray	tableaux extensibles des types de base (IloNum, IloInt, IloBool)
IloNumVarArray IloIntVarArray IloBoolVarArray	tableaux extensibles de variables (IloNumVar, IloIntVar, IloBoolVar)
IloRangeArray	tableaux extensibles de contraintes
IloAdd	Macro pour définir des objets et les ajouter au modèle d'un coup

- Résolution:

IloCplex	résoudre le modèle avec l'aide de CPLEX
-----------------	---

Classes de base:

- **IloExtractable** : classe mère pour les classes de construction du modèle

end()	détruit l'élément (remplace le destructeur)
getEnv()	récupère l'environnement auquel l'objet est rattaché
removeFromAll()	enlève l'élément de tous les autres objets où il était lié
setName(const char*)	permet de donner un nom à l'élément
getObject()	récupérer "l'objet" assigné à l'élément
setObject(IloAny)	assigner un "objet" à l'élément

- **IloAlgorithm** : classe mère pour les classes de résolution de modèle

clear()	permet de réinitialiser le solveur
end()	détruit le solveur (remplace le destructeur)
extract(const IloModel)	permet de charger un modèle dans le solveur
setError(ostream&) setWarning(ostream&) setOut(ostream&)	permettent de rediriger les canaux d'erreur, d'avertissement et d'output pour le solveur

Environnement et modèle:

- **IloEnv** : définir l'environnement pour créer et résoudre un modèle

end()	détruit l'environnement (remplace le destructeur) et TOUS les objets rattachés
setError(ostream&) setWarning(ostream&) setOut(ostream&)	permettent de rediriger les canaux d'erreur, d'avertissement et d'output pour l'environnement
getMemoryUsage()	taille utilisée (en octets) du monceau
getTotalMemoryUsage()	taille allouée (en octets) du monceau

ex.: **IloEnv** env;

- **IloModel** : modéliser LP, MILP, QP, MIQP (variables, contraintes, objectif, etc.)

add (const IloExtractableArray &X) add (const IloExtractable &X)	ajoute l'élément ou le tableau X au modèle
remove (const IloExtractableArray &X) remove (const IloExtractable &X)	enlève l'élément ou le tableau X du modèle
<i>méthodes héritées</i>	<i>voir IloExtractable</i>

==> à ajouter au modèle : variables, contraintes, objectif

ex.: **IloModel** model(env);
model.add(obj);

Objectif:

- **IloObjective**: définir une fonction objectif

setConstant (IloNum)	initialiser la constante de l'objectif
setLinearCoef (variable, valeur) setLinearCoefs (variables, valeurs)	modifier un ou plusieurs coefficients de l'objectif
setExpr (expression)	modifier l'expression de l'objectif
setSense (IloObjective::Sense)	modifier la direction de l'optimisation
<i>méthodes héritées</i>	<i>voir IloExtractable</i>

ex.: [ilobendersatsp.cpp](#)

IloObjective obj(env);
obj.setSense(IloObjective::Minimize);
obj.setExpr(exp);

N.B.: *IloObjective::Sense* = *IloObjective::Minimize* ou *IloObjective::Maximize*

- **IloMinimize**, **IloMaximize** : donner la direction d'optimisation

ex.: **IloObjective** obj = **IloMinimize**(env);

Variables:

- **IloNumColumn** : définir des variables en ajoutant les coefficients un à un.

clear()	réinitialiser l'objet
operator +=()	ajouter des coefficients à la variable en construction
end()	détruit l'objet

N.B.: UTILISER LA MÉTHODE *end()* UNE FOIS QUE LA VARIABLE A ÉTÉ DÉFINIE AFIN D'ÉVITER LES FUITES DE MÉMOIRE

- **IloNumVar, IloIntVar, IloBoolVar** : déclarer des variables continues, entières ou binaires par défaut

setLB(valeur) setUB(valeur) setBounds(valeur, valeur)	modifier les bornes de la variable (inférieure, supérieure ou les deux)
<i>méthodes héritées</i>	<i>voir IloExtractable</i>

```
ex.: ilodiet.cpp      IloNumVarArray buy(env);
                        for (j = 0; j < n; j++) {
                            IloNumColumn col = cost(foodCost[j]);
                            for (i = 0; i < m; i++)
                                col += range[i](nutrPer[i][j]);
                            Buy.add(IloNumVar(col, foodMin[j], foodMax[j], type));
                            col.end();
                        }
```

- **IloConversion**: convertir le type des variables de type IloNumVar

end()	met fin à la conversion (REVENIR AU TYPE INITIAL)
<i>méthodes héritées</i>	<i>voir IloExtractable</i>

N.B.: On ne peut pas ajouter 2 conversions successives!!!

```
ex.:      IloConversion conv(env, varX, ILOINT);
          model.add(conv);
          model.remove(conv);
          conv.end();
```

Contraintes:

- **IloExpr**: construire une expression (contrainte)

setConstant (IloNum)	initialiser la constante de l'expression
setLinearCoef (<i>variable, valeur</i>) setLinearCoefs (<i>variables, valeurs</i>)	modifier un ou plusieurs coefficients de l'expression
operator += (<i>variable / expression / valeur</i>)	ajouter un élément à l'expression avec un coefficient positif
operator -= (<i>variable / expression / valeur</i>)	ajouter un élément à l'expression avec un coefficient négatif
<i>méthodes héritées</i>	<i>voir IloExtractable</i>

N.B.: UTILISER LA MÉTHODE *end()* UNE FOIS QUE LA CONTRAINTE (*IloRange*) A ÉTÉ DÉFINIE AFIN D'ÉVITER LES FUITES DE MÉMOIRE

- **IloRange**: déclarer des contraintes

setLB (<i>valeur</i>) setUB (<i>valeur</i>) setBounds (<i>valeur, valeur</i>)	modifier les bornes de la contrainte (inférieure, supérieure ou les deux)
setLinearCoef (<i>variable, valeur</i>) setLinearCoefs (<i>variables, valeurs</i>)	modifier un ou plusieurs coefficients de la contrainte
setExpr (<i>expression</i>)	modifier l'expression attachée à la contrainte (LHS)
<i>méthodes héritées</i>	<i>voir IloExtractable</i>

ex.: *facility.cpp*

```
for(j = 0 ; j < nbLocations ; j++){  
    IloExpr v(env);  
    for(i = 0; i < nbClients; i++){  
        v += supply[i][j];  
        model.add(v <= capacity[j] * open[j]);  
        // ou model.add(IloRange(env, 0, capacity[j] * open[j] - v, IloInfinity);  
        v.end();  
    }  
}
```


- **IloSum**(*variables*) : faire la somme d'un tableau de variables d'un seul coup

ex.: [facility.cpp](#)

```
IloArray<IloNumVarArray> supply(env, nbClients);

for(i = 0; i < nbClients; i++)
    supply[i] = IloNumVarArray(env, nbLocations, 0, 1, ILOINT);
for(i = 0; i < nbClients; i++)
    model.add(IloSum(supply[i]) == 1); //  $\sum (j) \text{supply}[i][j] == 1$ 
```

N.B.: IloSum ne peut s'appliquer que sur la DERNIÈRE dimension d'une matrice
 Dans le cas contraire, il faut se servir de **IloExpr** pour bâtir la contrainte.

- **IloScalProd**(*variables, valeurs*): faire le produit scalaire d'un tableau de variables et d'un tableau de valeurs de tailles identiques

ex.: [facility.cpp](#)

```
IloNumArray fixedCost(env);
IloNumVarArray open(env, nbLocations, 0, 1, ILOINT);
IloArray<IloNumVarArray> supply(env, nbClients);
IloExpr obj = IloScalProd(fixedCost, open); //  $\sum (j) \text{fixedCost}[j] * \text{open}[j]$ 

for(i = 0; i < nbClients; i++)
    obj += IloScalProd(cost[i], supply[i]); //  $\sum (i,j) \text{supply}[i][j] * \text{cost}[i][j]$ 
```

- **IloIfThen**(*condition If, condition Then*) : modéliser des contraintes de type "Big M" ou logiques

ex.: $x \leq M * y \implies$ **IloIfThen**(env, y == 0, x == 0);

ex.: [foodmanufact.cpp](#)

Si les produits p1 ou p2 sont utilisés (+ de 20), alors le produit p3 le sera aussi:

```
model.add(IloIfThen(env, (use[p1] >= 20) || (use[p2] >= 20), use[p3] >= 20));
```

Tableaux + macro:

- **IloAdd** : Macro qui permet de définir des objets et de les ajouter au modèle en même temps

ex.: ilodiet.cpp

```
IloObjective cost = IloAdd(mod, IloMinimize(env));  
// objectif créé et ajouté au modèle
```

==> est équivalent à:

```
IloObjective cost = IloMinimize(env);  
mod.add(cost);
```

- **IloArray** : Patron de classe pour créer des tableaux extensibles multi-dimensions

add(X)	ajouter X au tableau
operator[] (int i)	accéder à l'élément i du tableau
clear()	réinitialise le tableau
getSize()	retourne la taille du tableau
end()	détruit le tableau (remplace le destructeur)

ex.: facility.cpp

```
typedef IloArray<IloNumArray> FloatMatrix; // matrice 2D de nombres  
typedef IloArray<IloNumVarArray> NumVarMatrix; // matrice 2D de variables
```

- **IloExtractableArray** : classe qui permet l'utilisation de tableaux extensibles pour définir des variables ou des contraintes à une ou plusieurs dimensions

end()	détruit tous les éléments du tableau (remplace le destructeur)
removeFromAll()	enlève les éléments du tableau de tous les autres objets où il était référencé
setNames (const char*)	permet de donner des noms aux éléments du tableau

- **IloNumArray, IloIntArray, IloBoolArray** : tableaux extensibles de nombres

contains (valeur)	cherche la valeur dans le tableau
méthodes héritées	voir <i>IloExtractableArray</i> et <i>IloArray</i>

- **IloNumVarArray, IloIntVarArray, IloBoolVarArray** : déclarer des tableaux de variables continues, entières ou binaires

add (variable) add (variables)	ajouter une ou plusieurs variables au tableau
setBounds (valeurs, valeurs)	modifier les bornes des variables du tableau
méthodes héritées	voir <i>IloExtractableArray</i> et <i>IloArray</i>

ex.: [*facility.cpp*](#)

```
typedef IloArray<IloNumVarArray> NumVarMatrix;
// matrice 2D de variables

NumVarMatrix supply(env, nbClients);
for(i = 0; i < nbClients; i++)
    supply[i] = IloNumVarArray(env, nbLocations, 0, 1, ILOINT);
```

ex.: [*cutstock.cpp*](#)

```
IloNumArray    newPatt(env, nWdth);
IloNumVarArray Use(env, nWdth, 0.0, IloInfinity, ILOINT);
patSolver.getValues(newPatt, Use);
```

- **IloRangeArray**: déclarer des tableaux extensibles de contraintes

add (contrainte) add (contraintes)	ajouter une ou plusieurs contraintes au tableau
setBounds (valeurs, valeurs)	modifier les bornes des contraintes du tableau
méthodes héritées	voir <i>IloExtractableArray</i> et <i>IloArray</i>

```
ex.: IloRangeArray tab(env);
    tab.add(IloRange(env, 0.0, 100.0));
IloRangeArray range (env, nutrMin, nutrMax); ==> ilodiet.cpp
```

Classes relatives au solveur:

- **IloCplex** : résoudre un modèle avec l'aide de CPLEX

LP + MIP	solve()	lance la résolution du modèle (relaxation continue suivi de B&C)
	getCplexStatus()	retourne le statut de l'optimisation
	exportModel(char*)	exporte le modèle dans un fichier
	getObjValue()	retourne la valeur de l'objectif
	getValue(variable / expression) getValues(valeurs, variables)	récupère les valeurs des variables à la fin de l'optimisation
	setParam(paramètre, valeur)	modifier la valeur de paramètres
	tuneParam()	recherche de meilleures valeurs pour paramètres
LP	getDual(contrainte), getDuals(valeurs, variables / expressions)	récupère les valeurs duales de contraintes
	getReducedCost(variable) getReducedCosts(valeurs, variables)	récupère les coûts réduits de variables
	getSlack(contrainte), getSlacks(valeurs, contraintes)	récupère les valeurs des surplus/écarts des contraintes
MIP	getBestObjValue()	retourne la meilleure borne restante
	addLazyConstraint(contrainte) addLazyConstraints(contraintes)	ajoute des contraintes LAZY dans un pool de contraintes (applicable sur les solutions entières)
	addUserCut(contrainte), addUserCuts(contraintes)	ajoute des contraintes USER au pool de contraintes (applicable sur les solutions fractionnaires)
	solveFixed()	résoudre le modèle avec variables fixées (accès aux duals, slacks, etc.)
	addMIPStart(variables, valeurs)	point de départ pour résoudre le MIP
	setPriority(variable, valeur) setPriorities(variables, valeurs)	permet de donner la priorité aux variables lors du branchement
	use(IloCplex::Callback)	permet l'utilisation de callbacks pour intervenir pendant le B&C
	<i>méthodes héritées</i>	<i>voir IloAlgorithm</i>

Paramètres utiles:

IloCplex::Param::	<i>(préfixe)</i>
Threads	limite sur le nombre de threads utilisés (par défaut: TOUS les CPUs) SVP: utiliser 1
Advance	réoptimisation
TimeLimit	temps maximum d'optimisation (en secondes)
ClockType	type de temps utilisé (<i>default</i> : horloge)
Preprocessing::Presolve	appliquer ou non presolve
Preprocessing::Symmetry	briser la symétrie du modèle (MIP)
Preprocessing::BoundStrength	tenter de fixer des variables (MIP)
Simplex::Tolerances::Optimality	tolérance sur l'optimalité de la solution
Simplex::Tolerances::Feasibility	tolérance sur la réalisabilité des variables
Simplex::Display	affichage de l'optimisation par simplexe
Emphasis::MIP	emphase du type de recherche (OPT vs FEAS)
Emphasis::Numerical	précision numérique
Emphasis::Memory	conserver mémoire si possible
RootAlgorithm	algorithme utilisé pour la relaxation (racine)
NodeAlgorithm	algorithme utilisé pour les autres noeuds
MIP::Display	affichage du B&C
MIP::Strategy::File	créer un fichier de stockage de noeuds
WorkMem	limite sur la mémoire utilisée avant stockage
MIP::Strategy::VariableSelect	sélection de la variable de branchement
MIP::Strategy::NodeSelect	sélection du type de recherche
MIP::Strategy::Branch	sélection de la direction initiale de branchement
MIP::Tolerances::LowerCutoff	écarte solutions ayant valeur < seuil
MIP::Tolerances::UpperCutoff	écarte solutions ayant valeur > seuil
MIP::Tolerances::MIPGap	gap limite (en %)
MIP::Tolerances::Integrality	tolérance sur l'intégralité des variables
MIP::Limits::Solutions	limite sur le nombre de solutions entières
MIP::Limits::Nodes	limite sur le nombre de noeuds évalués
MIP::Limits::TreeMemory	limite sur la taille de l'arbre de branchement

Callbacks (rappels d'optimisation): (OPTIONNEL)

==> permettent d'intervenir pendant le branch&cut.

- Types de callbacks:

information	accéder à des informations sur l'optimisation en cours sans interférer dans la recherche
diagnostic	surveiller une optimisation et éventuellement la terminer
contrôle	contrôler la recherche pendant la procédure de B&C et ainsi intervenir directement dans le processus

- Callbacks de contrôle:

UserCutCallback	ajouter des coupes aux noeuds fractionnaires
LazyConstraintCallback	ajouter des coupes aux noeuds entiers
NodeCallback	interroger (et remplacer) le noeud suivant traité par CPLEX lors de la recherche
SolveCallback	spécifier et configurer l'option du solveur utilisée pour la résolution de chaque noeud
HeuristicCallback	implanter une heuristique pour trouver une meilleure solution entière aux noeuds
BranchCallback	interroger (et remplacer) la méthode de branchement utilisée à chaque noeud
IncumbentCallback	consulter et éventuellement rejeter des "fausses" solutions entières

- Macros à utiliser pour définir les callbacks (LAZY / USER):

ILOLAZYCONSTRAINTCALLBACK0 (name)
ILOLAZYCONSTRAINTCALLBACK1 (name, type1, x1)
ILOLAZYCONSTRAINTCALLBACK2 (name, type1, x1, type2, x2)
...
ILOLAZYCONSTRAINTCALLBACK7 (name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)
ILOUSERCUTCALLBACK0 (name)
ILOUSERCUTCALLBACK1 (name, type1, x1)
ILOUSERCUTCALLBACK2 (name, type1, x1, type2, x2)
...
ILOUSERCUTCALLBACK7 (name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

ex.: **ILOUSERCUTCALLBACK3**(CtCallback, *IloExprArray*, *lhs*,
IloNumArray, *rhs*, *IloNum*, *eps*)

-> définit une fonction de type *UserCutCallback* ayant pour nom *CtCallback* recevant 3 paramètres (*lhs*, *rhs*, *eps*) de types respectifs (*IloExprArray*, *IloNumArray*, *IloNum*)

ex.: *cplex.use*(CtCallback(*env*, *lhs*, *rhs*, *1e-05*);

-> demande au solveur d'utiliser le *UserCutCallback* ayant pour nom *CtCallback* pendant l'optimisation avec les paramètres: *lhs*, *rhs*, *1e-05*

LAZY + USER	add (<i>contrainte</i>)	ajoute une coupe globale au modèle
	addLocal (<i>contrainte</i>)	ajoute une coupe locale au modèle
	<i>méthodes héritées</i>	<i>voir tableau plus loin</i>
USER	abortCutLoop ()	sort de la phase de coupes et retourne au branchement
	isAfterCutLoop ()	retourne <i>IloTrue</i> si callback appelé une dernière fois après que la dernière coupe a été générée (<i>IloFalse</i> sinon)

- Méthodes héritées (communes à *LazyConstraintCallback* et *UserCutCallback*)

getLB (<i>variable</i>) getLBs (<i>valeurs, variables</i>) getUB (<i>variable</i>) getUBs (<i>valeurs, variables</i>)	récupère une ou plusieurs bornes inférieures/supérieures des variables du modèle
getNodeId ()	récupère le ID unique du noeud courant
getObjValue ()	récupère la valeur de l'objectif du noeud courant
getSlack (<i>contrainte</i>) getSlacks (<i>contraintes</i>)	récupère les valeurs des surplus/écarts des contraintes au noeud courant
getValue (<i>variable / expression</i>) getValues (<i>valeurs, variables</i>)	récupère les valeurs des variables (expression) au noeud courant
getBestObjValue ()	récupère la meilleure borne fractionnaire de tous les noeuds restants
getIncumbentObjValue ()	récupère la valeur de l'objectif de la meilleure solution entière trouvée
getIncumbentValue (<i>variable / expression</i>) getIncumbentValues (<i>valeurs, variables</i>)	récupère les valeurs des variables (expression) tirées de la meilleure solution entière trouvée
getNodeNodes ()	récupère le nombre de noeuds évalués
getNodeRemainingNodes ()	récupère le nombre de noeuds restants
hasIncumbent ()	retourne <code>IloTrue</code> si une solution entière a été trouvée (<code>IloFalse</code> sinon)
getModel ()	récupère le modèle courant
abort ()	met fin à l'optimisation
getEnv ()	récupère l'environnement courant

N.B.: À l'intérieur de la fonction (macro) définie par l'utilisateur, le seul moyen de récupérer les valeurs des variables/contraintes au noeud courant est de passer les tableaux correspondants en paramètre comme dans l'exemple suivant:

ex.: [ilobendersatsp.cpp](#)

```
ILOUSERCUTCALLBACK5(BendersUserCallback, IloArray<IloIntVarArray>, x,  
                    IloCplex, workerCplex, IloNumVarArray, v,  
                    IloNumVarArray, u, IloObjective, workerObj)  
{  
    // Skip the separation if not at the end of the cut loop  
  
    if ( !isAfterCutLoop() )    return;  
  
    IloInt i;  
    IloEnv masterEnv = getEnv();  
    IloInt numNodes = x.getSize();  
  
    // Get the current x solution  
  
    IloArray<IloNumArray> xSol(masterEnv, numNodes);  
    for (i = 0; i < numNodes; ++i) {  
        xSol[i] = IloNumArray(masterEnv);  
        getValues(xSol[i], x[i]);  
    }  
  
    // Benders' cut separation  
  
    IloExpr cutLhs(masterEnv);    // expression de la coupe  
    IloNum cutRhs;  
  
    IloBool sepStat = separate(x, xSol, workerCplex, v, u, workerObj, cutLhs, cutRhs);  
    if ( sepStat ) {  
        add(cutLhs >= cutRhs).end();  
        // crée la coupe, l'ajoute au modèle et détruit l'objet temporaire créé  
    }  
  
    // Free memory  
  
    cutLhs.end();  
    for (i = 0; i < numNodes; ++i)  
        xSol[i].end();  
    xSol.end();  
    return;  
}
```