Programmation d'un *Branch & Cut* en C++ avec **Cplex**

Hélène Toussaint, avril 2021

Ce document complète le tutoriel [1] de présentation générale de la librairie **Cplex** *Concert Technology* (utilisation de **Cplex** dans un programme C++) en présentant des notions plus avancées : génération dynamique de coupes et utilisation de *callbacks* pour programmer un *Branch & Cut.*

But de ce document : Apprendre à générer des coupes dynamiquement et à programmer un *Branch & Cut* à l'aide de la bibliothèque **Cplex** *Concert Technology* (C++). On se focalise ici sur la partie pratique, on suppose la partie théorique connue du lecteur même si quelques notions théoriques sont brièvement rappelées.

Prérequis : C++, **Cplex** *Concert Technology* pour C++ (savoir construire un modèle : voir [1], [2]), programmation linéaire (voir [3] pour un cours détaillé).

Sommaire

1 AJOUT DE COUPES DANS UN PL

Dans cette section on se place dans le cadre de la résolution de programmes linéaires (*i.e*. uniquement en variables fractionnaires) et on s'intéresse à l'ajout de coupes dynamique.

1.1 RAPPEL DE LA METHODE

La génération de coupes est une méthode utilisée quand le modèle possède un très grand nombre de contraintes (par exemple exponentiel par rapport au nombre de variables).

Soit $\mathcal{P}$ un programme linéaire (aussi appelé *modèle* dans la terminologie **Cplex**) et $\mathcal{C}$ l'ensemble des contraintes de $\mathcal{P}$. L'idée est d'essayer de résoudre le modèle $\mathcal{P}$ sans être obligé d'y inclure explicitement toutes les contraintes. On forme alors le modèle initial $\mathcal{P}_0$ avec un sous-ensemble $\mathcal{C}_0 \subset \mathcal{C}$ des contraintes, on le résout et on examine la solution. Si la solution est valide (*i.e.* toutes les contraintes de $\mathcal{C}$ sont respectées) alors on a la solution optimale. Sinon on ajoute au modèle les contraintes dans $\mathcal{C}$ violées par la solution. On recommence itérativement ce processus jusqu'à obtenir une solution qui respecte toutes les contraintes. On espère atteindre cette solution avant d'avoir intégré explicitement toutes les contraintes au modèle.

Remarquons qu'on appelle *coupes* les contraintes ajoutées itérativement au modèle car elles *coupent* (*i.e.* rendent invalide) la solution courante.

1.2 EN PRATIQUE

En pratique on construit le modèle initial $\mathcal{P}_0$ à l'aide des fonctions habituelles de la *Concert Technology* (voir [1]). Le modèle est alors stocké en mémoire et pourra être modifié ultérieurement si nécessaire.

Après résolution, **Cplex** garde en mémoire certaines informations, obtenues pendant la résolution. En particulier, **Cplex** maintient à jour une base qui servira de point de départ pour reprendre les itérations du simplexe là où il s'était arrêté.

Pour que **Cplex** tire parti au mieux de la résolution précédente, il est conseillé d'utiliser le simplexe dual pour résoudre à nouveau le modèle, après avoir ajouté une ou plusieurs contraintes :

```
cplex.setParam(IloCplex::Param::RootAlgorithm, IloCplex::Dual);
```

1.3 EXEMPLE

On montre ici comment ajouter des contraintes à un modèle déjà résolu et relancer la résolution après modification du modèle. On utilise l'exemple **`ilolpex3.cpp`** (fourni dans le répertoire d'installation de **Cplex**).

La construction du modèle initial et l'accès à la solution ne sont pas détaillés (voir [1] pour apprendre à construire un PL via la *Concert Technology*). On se focalise sur l'ajout des coupes et la « réoptimisation » du modèle.

```
// construction du modèle initial
// [...]

//resolution du modèle initial
cplex.solve();

//ajout des coupes au modèle (le modèle étant déjà associé à une instance
//d'IloCplex, les changements seront bien pris en compte lors de la prochaine optimisation)
//exemple de coupes :
model.add(2*x[10] + 5*x[11] == 2);
model.add(  x[0] + x[2] + x[5] == 3);

//relance l'optimisation avec le simplexe dual (c'est en général l'algorithme qui tire le
plus de bénéfice de la base optimale trouvée par l'optimisation précédente)
cplex.setParam(IloCplex::Param::RootAlgorithm, IloCplex::Dual);
cplex.solve();

// accès a la solution et ajout éventuel d'autres coupes
// [...]
```

Exemple : ajout de coupes et nouvelle résolution du modèle modifié

Remarque

Cet exemple est très basique car on ne modifie le modèle qu'une seule fois. Ce qui est important de comprendre ici c'est qu'on peut facilement modifier le modèle après l'avoir résolu et relancer l'optimisation. On remarque que les fonctions utilisées sont les mêmes que pour créer le modèle initial. La seule particularité est l'utilisation du simplexe dual pour tirer parti des informations stockées pendant la première optimisation.

On pourrait bien sûr modifier le modèle de manière itérative : à chaque itération on recherche des contraintes violées, on les ajoute, et on résout à nouveau avec le simplexe dual (c'est d'ailleurs le cas d'utilisation le plus classique). Dans ce cas, il est suffisant de paramétrer l'algorithme de résolution une seule fois (à la première itération) : dès lors qu'on modifie un paramètre avec la fonction **`cplex.setParam()`** il reste modifié tant qu'on ne met pas fin à l'environnement **Cplex** correspondant.

2 AJOUT DE COUPES DANS UN PLNE : *BRANCH & CUT*

Dans la première section on a vu comment ajouter très facilement de nouvelles contraintes (ou coupes) à un modèle en nombres fractionnaires et relancer la résolution.

Cependant l'ajout de coupes dynamique concerne généralement des programmes linéaires en nombres entiers ou mixtes. La fait d'avoir des variables entières complique considérablement la résolution du modèle et également la manière dont on va ajouter les coupes.

En effet, dès lors qu'il y a la présence de variables entières (y compris binaires) dans un modèle (même en très petit nombre), l'algorithme du simplexe n'est plus suffisant pour résoudre le modèle (car il ne sait pas prendre en compte les contraintes d'intégrité). On utilise alors une méthode par *séparation / évaluation* (plus connue sous le nom de *Branch & Bound*) qui consiste à explorer un arbre de recherche dans lequel chaque nœud est associé à un programme linéaire, résolu à l'aide du simplexe (voir [3] pour plus de détails). Ainsi la méthode **`solve()`** de Cplex, appelée sur un modèle qui contient des variables entières, lance la recherche arborescente dans son intégralité. Pour ajouter les coupes de manière efficace il est donc primordial d'interagir avec **Cplex** pendant la résolution, et de ne pas attendre la fin de la résolution comme on le fait dans le cas d'un modèle qui ne contient que des variables fractionnaires (section 1).

L'idée est donc d'interagir avec **Cplex** en ajoutant des contraintes (ou coupes) pendant le déroulement du *Branch & Bound. Un Branch & Bound* avec ajout de coupes dynamique s'appelle *Branch & Cut.*

2.1 LES TYPES DE COUPES DANS **CPLEX**

Avant de présenter les fonctionnalités de **Cplex** qui vont nous permettre de programmer un *Branch & Cut*, il est important de comprendre que **Cplex** distingue deux types de coupes ([5]) :

- les *users cuts,*
- les *lazy constraints.*

Les *users cuts* sont les coupes qui permettent d'améliorer la relaxation fractionnaire : elles ne sont pas indispensables au modèle, elles permettent simplement de le renforcer : n'importe quelle solution entière au problème vérifie implicitement les *users cuts.*

Les *lazy constraints* sont des contraintes indispensables au modèle mais que l'utilisateur choisit d'ajouter itérativement, uniquement lorsqu'elles sont violées par la solution courante du *Branch & Bound*. Les *lazy constraints* sont donc simplement un sous-ensemble des contraintes du modèle que l'utilisateur ne souhaite pas ajouter initialement au modèle (par exemple parce qu'elles sont trop nombreuses).

Remarque : dans les deux cas, seules les contraintes linéaires sont acceptées par **Cplex** (il n'est pas possible d'ajouter des contraintes quadratiques en tant que *user cut* ou *lazy contraints*).

Lors de la résolution, ces deux types de coupe ne sont pas traités de la même manière par **Cplex** :

- **Cplex** peut vérifier à n'importe quelle étape de la résolution la validité des *user cuts* mais il ne les vérifie pas forcément lorsqu'une solution entière est trouvée,

- les *lazy constraints* sont vérifiées :
    - soit lorsque **Cplex** a identifié une solution entière candidate (*i.e.* à un nœud de l'arbre le simplexe a fourni une solution entière : **Cplex** vérifie alors les *lazy constraints* et conserve la solution si elle n'en viole aucune),

- soit lorsque la relaxation est non bornée : **Cplex** applique alors les *lazy constraints* pour essayer de couper la direction non bornée.

2.2 *USERS CUTS*, *LAZY CONSTRAINTS* ET PRETRAITEMENT

Par défaut, **Cplex** applique un prétraitement ([6]) à tous les modèles (réduction du nombre de variables, de contraintes, déduction de la valeur de certaines variables etc…).
Il se peut que **Cplex** ne soit pas capable de prendre en compte certaines *user cuts* lors du prétraitement. Il affiche alors un message du type « *Could not crush* n *user cuts* », où n est le nombre de *user cuts* non prises en compte. Certains callbacks, liés aux *user cuts*, peuvent également afficher un *warning* indiquant que des *user cuts* n'ont pas été utilisées durant le prétraitement. Ces *warnings* ne sont pas critiques dans le sens où l'utilisation des *user cuts* est optionnelle : le fait de ne pas les prendre en compte ne remet pas en question la validité de la solution obtenue.

Si **Cplex** détecte des *lazy constraints* alors certains prétraitements ([7]) sont désactivés. En effet les *lazy constraints* sont indispensables au modèle (contrairement aux *user cuts*) donc les prétraitements qui ne peuvent pas prendre en compte les *lazy contraints* sont automatiquement désactivés.

L'ajout de *user cuts* et *lazy constraints* se fait par l'intermédiaire d'objets appelés *callbacks*.

2.3 QU'EST-CE QUE LES *CALLBACKS* ?

Dans **Cplex** les *callbacks* ([4]) permettent à l'utilisateur d'agir à des moments spécifiques durant le déroulement du *Branch & Bound*. Il existe deux types de *callbacks* :

- les *query callbacks* : ils permettent de récupérer de l'information à différentes étapes de la résolution (prétraitement, *Branch & Bound*, application de certains types de coupes implémentés nativement dans **Cplex**, …),

- les *control callbacks :* ils permettent d'agir sur la résolution en contrôlant certaines étapes du *Branch & Bound* (choix du prochain nœud, création des nœuds fils, ajout de coupes, …)

Ces *callbacks* ont une implémentation particulière et ne s'utilisent pas exactement comme des fonctions traditionnelles. Il s'agit d'ailleurs d'objets (au sens C++) et non de fonctions. Cependant on utilise souvent improprement le mot fonction pour désigner les *callbacks*.

Il existe des macros de la forme **`ILOXXXCALLBACK`**$\boldsymbol{n}$ (où n = 0, …, 7 est le nombre d'arguments du *callback* et **`XXX`** désigne le nom du *callback*) qui permettent de faciliter la création des callbacks par l'utilisateur. Ce sont ces macros qui sont utilisées dans les exemples fournis par **Cplex** et que nous utiliserons dans la suite de ce document également.

2.4 LES *CALLBACKS* POUR PROGRAMMER UN *BRANCH & CUT*

Dans le cadre de la programmation d'un *Branch & Cut*, deux callbacks vont nous être utiles :

- celui qui permet d'ajouter dynamiquement des *lazy constraints* et que l'on crée à l'aide de la macro ILOLAZYCONSTRAINTCALLBACKn ($n = 0, \dots, 7$) [8]

- celui qui permet d'ajouter dynamiquement des *user cuts* et que l'on crée à l'aide de la macro ILOUSERCUTCALLBACKn ($n = 0, \dots, 7$) [9]

Les macros ILOLAZYCONSTRAINTCALLBACKn et ILOUSERCUTCALLBACKn permettent d'utiliser les *callbacks* comme des fonctions. La syntaxe est la suivante :

```
ILOLAZYCONSTRAINTCALLBACKn (name, type1, x1, …, typen, xn)
```

ILOLAZYCONSTRAINTCALLBACKn est le nom de la macro qui crée le *callback* permettant d'inclure dynamiquement des *lazy callbacks* à la résolution. Elle doit être écrite exactement comme cela en remplaçant $\boldsymbol{n}$ par le nombre de paramètres (on peut avoir jusqu'à 7 paramètres, pas plus). Entre parenthèses il faut fournir les informations qui vont permettre à cette macro de créer le *callback* :

- tout d'abord son nom (**name**) : tout comme pour une fonction on peut choisir le nom que l'on souhaite,
- ensuite la liste des paramètres séparés par des virgules : on donne d'abord le type puis le nom de la variable. Contrairement à une fonction classique le type et le nom de la variable sont séparés par une virgule. Les paramètres sont les données nécessaires à l'écriture des contraintes (typiquement les variables du programme linéaire et les données d'entrée nécessaires à l'écriture de la contrainte).

Un exemple d'implémentation de *callback* est donné section 2.5.1.

Le fait de créer un *callback* ne l'inclut pas automatiquement à la résolution. Il faut l'appeler explicitement avant de lancer la résolution (voir section 2.5.4).

2.5 EXEMPLE COMPLET

On propose dans cette section d'illustrer l'utilisation des *callbacks* sur un exemple simple de résolution d'un programme linéaire en nombres entiers par *Branch & Cut.* On détaille ici chaque étape, le programme complet est donné en annexe.

On considère le PLNE ci-dessous, formé de N variables entières $x_0, \dots, x_{N-1}$. L'objectif est de minimiser une somme pondérée des x_i (où p_i est le coefficient de pondération de x_i) tout en imposant que la somme des variables soit au moins égale à $\frac{N}{4}$ ($(C1)$) et que la somme de chaque couple (x_i, x_j) de variables ne dépasse pas un coefficient a_{ij} donné ($(C2)$).

Minimiser $\sum_{i=0}^{N-1} p_i x_i$

sous

$$\sum_{i=0}^{N-1} x_i \geq \frac{N}{4} \qquad (C1)$$

$$\forall (i,j) \in [\![0, N-1]\!]^2, j \neq i, \quad x_i + x_j \leq a_{ij} \qquad (C2)$$

$$\forall i \in [\![0, N-1]\!], \quad x_i \in \mathbb{N}$$

Il est clair que ce PLNE est suffisamment simple pour être programmé directement (sans les *callbacks*). Néanmoins, on suppose que l'on souhaite créer dynamiquement l'ensemble des contraintes (2).

2.5.1 DEFINITION ET IMPLEMENTATION DES *CALLBACKS*

Dans cette section on montre comment programmer les deux *callbacks* qui vont permettre d'ajouter les contraintes (2) de manière dynamique (et donc de mettre en œuvre un *Branch & Cut*).

Contrairement à ce qui est habituel en C++, la définition et l'implémentation des *callbacks* se fait dans un unique fichier d'entête (**.h**). Ceci est dû au fait que les *callbacks* s'écrivent à l'aide de macros (il est possible de ne pas passer par les macros mais elles facilitent l'écriture des *callbacks*).

La contrainte (2) de notre PLNE peut être implémentée dans un *lazy constraint callback* de la manière suivante :

```
ILOLAZYCONSTRAINTCALLBACK2(MyLazyConstraintCallback, IloNumVarArray, x, vector<vector<int>>
&, a)
{
	cout << "   +++ LAZY CONSTRAINT CALLBACK  +++ " << endl;
	IloInt N = x.getSize();

	// 1. on recupere l'environnement courant
	IloEnv env = getEnv();

	// 2. on recupère la solution courante
	IloNumArray xSol(env, N);
	getValues(xSol, x);

	//3. on execute notre méthode de séparation : on cherche si xSol viole des
	     contraintes et on ajoute les contraintes violées
	for (int i = 0; i < N; ++i)
	{
		for (int j = 0; j < N; ++j)
		{
			if (i != j && xSol[i] + xSol[j] > a[i][j] + epsilon)
				add(x[i] + x[j] <= a[i][j]).end();
		}
	}

	//4. on libere la mémoire
```

```
        xSol.end();

        return;
}
```

On peut construire un *user cut callback* exactement de la même manière.

2.5.2 CREATION DU MODELE INITIAL

Dans un fichier séparé de celui implémentant les *callbacks*, on crée modèle avec la contrainte (1) uniquement.

```
        //declaration de l'environnement Cplex
        IloEnv env;

        //-----------------------------------------
        // 0. initialisation des données

        int N = 20;
        IloNumArray p(env, N), ones(env, N);
        vector< vector<int> > a(N, vector<int>(N));
        initInput(N, p, a);
        for (int i = 0; i < N; ++i)
                ones[i] = 1.0;

        //-----------------------------------------
        // 1. creation du modele initial

        IloModel model(env, "myModel");

        //declaration des variables
        IloNumVarArray x(env, N, 0, IloInfinity, IloNumVar::Int);


        //ajout des contraintes de type (1)
        model.add(IloScalProd(ones, x) >= N/4);


        //ajout de la fonction objectif
        model.add(IloMinimize(env, IloScalProd(p, x)));

        //on affiche le modèle dans un fichier pour vérification
        IloCplex cplex(model);
        cplex.exportModel("modInitial.lp");
```

La fonction **`initInput(N, p, a)`** génère aléatoirement des valeurs pour le vecteur **`p`** et la matrice **`a`**. Le vecteur **`ones`** sert simplement à faciliter l'écriture de la contrainte (1).

2.5.3 PARAMETRAGE DE **CPLEX** POUR LE *BRANCH & CUT*

Lorsqu'on utilise des callbacks, et notamment des *control callbacks* qui sont susceptibles de modifier le modèle durant le déroulement du *Branch & Cut*, quelques précautions doivent être prises :

- désactiver le prétraitement :

```
cplex.setParam(IloCplex::Param::Preprocessing::Presolve, IloFalse);
```

- fixer le nombre de threads à 1 :

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

- utiliser une recherche arborescente “traditionnelle”

```
cplex.setParam(IloCplex::Param::MIP::Strategy::Search, IloCplex::Traditional);
```

Remarque :
Cplex fixe déjà automatiquement certains paramètres lorsque des *control callbacks* sont utilisés. Par exemple le nombre de threads est automatiquement fixé à 1 car la modification de données globales au modèle peut créer des conflits entre les threads. Il est quand même préférable d’imposer nous même le paramétrage : pour mémoire et au cas où des changements auraient lieu entre différentes versions de **Cplex**.

2.5.4 INTEGRATION DES *CALLBACKS*

Avant d’appeler la méthode **`solve()`** de **Cplex**, il faut informer **Cplex** de l’utilisation des callbacks afin qu’ils soient bien pris en compte durant le déroulement du *Branch & Bound*. Ceci se fait à l’aide de la méthode **`use()`**. On suppose ici qu’on a deux *callbacks* qui prennent chacun en paramètre l’environnement **Cplex `env`**, le vecteur des variables **`x`** et la matrice de coefficients **`a`** :

```
	cplex.use(MyLazyConstraintCallback(env, x, a));
	cplex.use(MyUserCutCallback(env, x, a));
```

2.5.5 RESOLUTION ET AFFICHAGE DES RESULTATS

L’appel au solveur ainsi que l’accès à la solution se font de manière classique.

```
	// 5. affichage  de la solution
	if (cplex.getStatus() == IloAlgorithm::Optimal)
	{
		cout << "val. objectif : " << cplex.getObjValue() << endl;
		for (int i = 0; i < N; ++i)
			cout << "x[" << i << "] = " << cplex.getValue(x[i]) << endl;
	}
	else
		cout << "PAS DE SOLUTION OPTIMALE" << endl;
```

2.5.6 LIBERATION DE LA MEMOIRE

A la fin du programme, on appelle la méthode **`end()`** sur la variable d’environnement pour libérer la mémoire.

```
	env.end();
```

Le programme complet est disponible en annexe et peut être téléchargé à l’adresse https://perso.limos.fr/~hetoussa/doc/code_B&C.zip

2.6 EXEMPLE DE *BRANCH & CUT* FOURNI PAR **CPLEX**

L'exemple précédent montre toutes les étapes nécessaires à la programmation d'un *Branch & Cut* ainsi que la manière dont on implémente les *callbacks*. Cependant, c'est un exemple volontairement très simple, et qui ne permet donc pas de distinguer la nuance entre les *lazy constraints* et les *users cuts*. Pour cela il existe un exemple, disponible dans le répertoire d'installation de Cplex : **`ilobendersatsp.cpp`**

Cet exemple résout un TPS asymétrique et implémente deux méthodes de séparation distinctes : une pour les solutions fractionnaires et une pour les solutions entières.

3 REFERENCES

[1] Hélène Toussaint, Programmer en C++ avec Cplex : Concert Technology - Les bases, 2015. https://perso.limos.fr/~hetoussa/doc/ilocplex.pdf.

[2] IBM, IBM Knowledge Center, Tutoriel Concert Technology pour les utilisateurs du C++. https://www.ibm.com/support/knowledgecenter/fr/SSSA5P_12.10.0/ilog.odms.cplex.help/CPLEX/GettingStarted/topics/tutorials/Cplusplus/cpp_synopsis.html

[3] Pierre Fouilhoux, Programmation mathématique Discrète et Modèles Linéaires, Université Pierre et Marie Curie, Master IAD Module PDML, 2013. http://www-desir.lip6.fr/~fouilhoux/documents/PDML_poly.pdf

[4] IBM, IBM Knowledge Center, Generic callbacks. https://www.ibm.com/support/knowledgecenter/en/SS9UKU_12.10.0/com.ibm.cplex.zos.help/CPLEX/UsrMan/topics/progr_adv/callbacks/introCallbacks.html

[5] IBM, IBM Knowledge Center, What are user cuts and lazy constraints? https://www.ibm.com/support/knowledgecenter/en/SS9UKU_12.10.0/com.ibm.cplex.zos.help/CPLEX/UsrMan/topics/progr_adv/usr_cut_lazy_constr/02_defn.html

[6] IBM, IBM Knowledge Center, Advanced presolve routines. https://www.ibm.com/support/knowledgecenter/SSSA5P_12.10.0/ilog.odms.cplex.help/CPLEX/UsrMan/topics/progr_adv/presolve_adv/01_presolve_title_synopsis.html

[7] IBM, IBM Knowledge Center, Limitations on user-cut pools. https://www.ibm.com/support/knowledgecenter/en/SS9UKU_12.10.0/com.ibm.cplex.zos.help/CPLEX/UsrMan/topics/progr_adv/usr_cut_lazy_constr/05_limitations.html

[8] IBM, IBM Knowledge Center, Macro ILOLAZYCONSTRAINTCALLBACK0. https://www.ibm.com/support/knowledgecenter/fr/SSSA5P_12.10.0/ilog.odms.ide.help/refcppopl/html/macros/ILOLAZYCONSTRAINTCALLBACK0.html

[9] IBM, IBM Knowledge Center, Macro ILOUSERCUTCALLBACK0. https://www.ibm.com/support/knowledgecenter/fr/SSSA5P_12.10.0/ilog.odms.ide.help/refcppopl/html/macros/ILOUSERCUTCALLBACK0.html

4 ANNEXE

Le programme complet de la section 2.5 se divise en quatre fichiers :

- **main.cpp**
- **modele.cpp** : implémentation du modèle de base, résolution, affichage des résultats
- **modele.h** : définition des fonctions de **modele.cpp**
- **callback.h** : définition et implémentation des *callbacks*

```cpp
#include "Modele.h"

int main(int argc, char **argv)
{
	//fonction principale :
	//construit le modele et lance la resolution
	construitResout();


	return 0;
}
```

Fichier `main.cpp`

```cpp
#pragma once

#include <ilcplex/ilocplex.h>
#include <string>
ILOSTLBEGIN

#include <vector>

//initialisation des données d'entrée
void initInput(int N, IloNumArray & p,  vector<vector<int>> & a);

//construction du modèle initial et résolution par Branch & Cut
void construitResout();
```

Fichier `modele.h`

```cpp
#include "Modele.h"
#include "callback.h"


//initialisation des données d'entrée
void initInput(int N, IloNumArray & p, vector<vector<int>> & a)
{
	srand(874);
```

```
	for (int i = 0; i < N; ++i)
	{
		p[i] = rand() % 100;
		for (int j = i + 1; j < N; ++j)
			a[i][j] = a[j][i] = rand() % 2 + 1;
	}
}

//construction du modèle initial et résolution par Branch & Cut
void construitResout()
{
	//declaration de l'environnement Cplex
	IloEnv env;

	//----------------------------------------
	// 0. initialisation des données
	int N = 20;
	IloNumArray p(env, N), ones(env, N);
	vector< vector<int> > a(N, vector<int>(N));
	initInput(N, p, a);
	for (int i = 0; i < N; ++i)
		ones[i] = 1.0;

	//----------------------------------------
	// 1. creation du modele initial

	IloModel model(env, "myModel");

	//declaration des variables
	IloNumVarArray x(env, N, 0, IloInfinity, IloNumVar::Int);


	//ajout des contraintes de type (1)
	model.add(IloScalProd(ones, x) >= N/4);


	//ajout de la fonction objectif
	model.add(IloMinimize(env, IloScalProd(p, x)));

	//on affiche le modèle dans un fichier pour vérification
	IloCplex cplex(model);
	cplex.exportModel("modInitial.lp");


	//-------------------------------------------
	// 2. Resolution par Branch & Cut

	// 2.1 on fixe quelques paramètres nécessaires au fonctionnement du Branch & Cut
	cplex.setParam(IloCplex::Param::Preprocessing::Presolve, IloFalse);
	cplex.setParam(IloCplex::Param::Threads, 1);
	cplex.setParam(IloCplex::Param::MIP::Strategy::Search, IloCplex::Traditional);

	// 2.2 integration des callbacks a la resolution
	cplex.use(MyLazyConstraintCallback(env, x, a));
	cplex.use(MyUserCutCallback(env, x, a));

	// 2.3. resolution : les callbacks seront appelés automatiquement au moment opportun
	IloBool ok = cplex.solve();


	// 5. affichage  de la solution
	if (cplex.getStatus() == IloAlgorithm::Optimal)
	{
		cout << endl << "          ================================" << endl;
		cout << "val. objectif : " << cplex.getObjValue() << endl;
		for (int i = 0; i < N; ++i)
```

```
                        cout << "x[" << i << "] = " << cplex.getValue(x[i]) << endl;
        }
        else
                cout << "PAS DE SOLUTION OPTIMALE" << endl;

        //6. liberation de la memoire
        env.end();
}
```

Fichier `modele.cpp`

```
#pragma once


#include <ilcplex/ilocplex.h>
#include <string>
ILOSTLBEGIN

#include <vector>

const double epsilon = 0.000001;

//exemple d'un callback LazyConstraint pour séparer des solutions entières
ILOLAZYCONSTRAINTCALLBACK2(MyLazyConstraintCallback, IloNumVarArray, x, vector<vector<int>>
&, a)
{
        cout << "   +++ LAZY CONSTRAINT CALLBACK  +++ " << endl;
        IloInt N = x.getSize();

        // 1. on recupere l'environnement courant
        IloEnv env = getEnv();

        // 2. on recupère la solution courante
        IloNumArray xSol(env, N);
        getValues(xSol, x);

        //3. on execute notre méthode de séparation : on cherche si xSol viole des
contraintes et on ajoute les contraintes violées
        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N; ++j)
                {
                        if (i != j && xSol[i] + xSol[j] > a[i][j] + epsilon)
                                add(x[i] + x[j] <= a[i][j]).end();
                }
        }

        //4. on libere la mémoire
        xSol.end();

        return;

}


//exemple d'un callback UserCut pour séparer des solutions fractionnaires
// ici on a la même séparation que pour les solutions entières
ILOUSERCUTCALLBACK2(MyUserCutCallback, IloNumVarArray, x, vector<vector<int>> &, a)
{
        cout << "   +++ USER CUT CALLBACK  +++ " << endl;
        IloInt N = x.getSize();

        // 1. on recupere l'environnement courant
```

```
        IloEnv env = getEnv();

        // 2. on recupère la solution courante
        IloNumArray xSol(env, N);
        getValues(xSol, x);

        //3. on execute notre méthode de séparation : on cherche si xSol viole des
contraintes et on ajoute les contraintes violées
        for (int i = 0; i < N; ++i)
        {
                for (int j = 0; j < N; ++j)
                {
                        if (i != j && xSol[i] + xSol[j] > a[i][j] + epsilon)
                                add(x[i] + x[j] <= a[i][j]).end();
                }
        }

        //4. on libere la mémoire
        xSol.end();


        return;
}
```

Fichier `callback.h`