API CPLEX

CPLEX est un outil commercial d'optimisation mathématique. Il doit son nom à l'algorithme du simplexe, un algorithme relativement ancien pour les problèmes d'optimisation linéaire, mais il supporte actuellement bien d'autres types de programmes mathématiques (programmes entiers, mais également non linéaires – quadratiques). Il fait partie [des outils les plus performants à ce niveau](http://scip.zib.de/), de par sa rapidité de traitement.

Il dispose d'une interface interactive en ligne de commande (CPLEX Interactive Optimizer), mais également d'interfaces dans divers langages de programmation, dont C++ et Java (Concert Technology). Ce document se focalise sur ces interfaces.

[Une version d'évaluation de trois mois](http://www-01.ibm.com/software/websphere/products/optimization/cplex-studio-preview-edition/), grandement limitée dans la taille des problèmes, est disponible gratuitement ; pour le monde académique, [une licence gratuite sans limite](http://www-304.ibm.com/ibm/university/academic/pub/page/academic_initiative) est également proposée.

# Architecture d'une application

Une application utilisant CPLEX travaillera principalement avec un objet IloCplex ou IloEnv : il permet de créer des variables (IloNumVar) et des contraintes (IloConstraint), mais aussi d'appeler le solveur et de récupérer les solutions. Tous les objets créés ne seront que des interfaces vers la représentation interne du solveur CPLEX de ces variables et contraintes : le fonctionnement interne de CPLEX n'est pas montré, seule une version abstraite est disponible – et largement suffisante pour la plupart des besoins.

Une application se déroulera donc en trois phases :

* la création du modèle ;
* sa résolution ;
* l'utilisation de la solution.

Pour de petits modèles, il est envisageable de n'utiliser qu'une fonction qui gère l'entièreté du modèle, mais cette solution n'est pas applicable à grande échelle. Pour un modèle de taille plus conséquente, le paradigme objet, utilisé pour la conception de l'API Concert, se révélera souvent utile. Une analyse du problème à modéliser donne une liste de concepts qui y interviennent (par exemple, les ressources disponibles, les machines à utiliser, les objets à fabriquer) : chacun deviendra une classe, dont les instances seront responsables de la création des variables et des contraintes qui leur correspondent (une machine ne gérera que sa propre production).

# Création d'un modèle

La première étape est d'instancier CPLEX. En C++, il s'agira de créer un objet IloEnv, puis un modèle IloModel ; en Java, une instance de IloCplex gère l'entièreté du processus.

**Program listing. Language: c++.**

IloEnv env;  
IloModel model(env);

**Program listing. Language: java.**

IloCplex cplex = new IloCplex();

De là, il est possible de créer une variable, par exemple entière, qui prend ses valeurs entre zéro et quarante-deux, avec un nom "x" (il sera utile pour le débogage).

**Program listing. Language: c++.**

IloNumVar x(env, 0, 42, IloNumVar::Int, "x");

**Program listing. Language: java.**

IloNumVar x = cplex.intVar(0, 42, "x");

Il est également possible de créer des tableaux de variables, par exemple booléennes, dont les noms sont donnés par un tableau de chaînes de caractères (en Java).

**Program listing. Language: c++.**

IloNumVarArray y(env, 2, 0, 1, IloNumVar::Bool);

**Program listing. Language: java.**

String[] namesY = new String[] { "y1", "y2" };  
IloNumVar[] y = cplex.boolVarArray(2, namesY);

De là, l'API permet d'exprimer des contraintes entre ces variables, chacune avec un nom donné.

**Program listing. Language: c++.**

model.add(y[0] <= y[1], "c1");  
model.add(x <= 1000 \* y[1], "c2");

**Program listing. Language: java.**

cplex.addLe(y[0], y[1], "c1");  
cplex.addLe(y[0], cplex.prod(1000, y[1]), "c2");

La dernière étape est de créer un objectif.

**Program listing. Language: c++.**

model.add(IloMinimize(env, x));

**Program listing. Language: java.**

cplex.addMinimize(x);

# Optimisation d'un modèle

La résolution du modèle se réduit ensuite à l'appel d'une fonction.

**Program listing. Language: c++.**

IloCplex cplex(model);  
cplex.solve();

**Program listing. Language: java.**

cplex.solve();

La méthode solve() retourne un booléen vrai si l'optimisation s'est déroulée sans problème (si une solution a été trouvée).getStatus() donne plus de détail (solution optimale trouvée, solution faisable mais non optimale trouvée, modèle infaisable, etc.).

La valeur de l'objectif est accessible par la méthode getObjValue().

**Program listing.**

double objective = cplex.getObjValue();

La valeur d'une variable particulière s'obtient en passant son objet IloNumVar en paramètre à getValue().

**Program listing.**

double xV = cplex.getValue(x);

Pour récupérer les valeurs de plusieurs variables, getValues() prend en paramètre un tableau.

**Program listing. Language: c++.**

double yV[] = cplex.getValues(y);

**Program listing. Language: java.**

double[] yV = cplex.getValues(y);

# Résolution d'infaisabilité

En cas de problème dans la résolution (pas de solution faisable, loin de l'optimalité, etc.), il est bien souvent utile d'exporter le modèle créé. CPLEX permet de créer très facilement des fichiers LP contenant toutes les contraintes et objectifs, contenant notamment leur nom.

**Program listing.**

cplex.exportModel("model.lp");

Ce fichier peut alors être chargé dans l'interface interactive, où [un outil de détection de conflits](http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r6/index.jsp?topic=%2Filog.odms.ide.help%2Frefcppopl%2Fhtml%2Fconflict.html) peut se révéler précieux (il est également accessible par les API CPLEX).

Dans la console interactive, une fois le modèle chargé et optimisé, la commandeconflict détermine un sous-ensemble d'inégalités rendant le problème non faisable (IIS, pour irreducible infeasible set), qu'il présente grâce à display conflict all sous la forme d'un problème de faisabilité reprenant ces quelques contraintes (souvent assez peu).

CPLEX> read model.lp  
Problem 'Problems/model.lp' read.  
Read time = 0.02 sec. (0.55 ticks)  
  
CPLEX> optimize  
Row 'c4257' infeasible, all entries at implied bounds.  
Presolve time = 0.00 sec. (0.89 ticks)  
…  
  
CPLEX> conflict  
Refine conflict on 4884 members...  
  
 Iteration Max Members Min Members  
 1 4274 0  
 …  
 37 7 7  
  
Minimal conflict: 7 linear constraint(s)  
 0 lower bound(s)  
 0 upper bound(s)  
Conflict computation time = 0.22 sec. Iterations = 37  
Deterministic time = 38.98 ticks (177.99 ticks/sec)  
  
CPLEX> display conflict all  
Minimize  
 obj:  
Subject To  
 off\_t1\_th4: on\_t1\_th4 + off\_t1\_th4 = 1  
 off\_t2\_th4: on\_t2\_th4 + off\_t2\_th4 = 1  
 timeToStart\_t0\_th4: - starting\_t0\_th4 + off\_t1\_th4 >= 0  
 startConjunction\_t0\_th4: - on\_t2\_th4 + starting\_t0\_th4 >= 0  
 stopIfPreviousOn\_t0\_th4: stopping\_t0\_th4 <= 0  
 stopConjunction\_t2\_th4: - on\_t1\_th4 - off\_t2\_th4 + stopping\_t0\_th4 >= -1  
 c4225: - on\_t1\_th4 <= -1  
\Sum of equality rows in the conflict:  
\ sum\_eq: on\_t1\_th4 + on\_t2\_th4 + off\_t1\_th4 + off\_t2\_th4 = 2  
Bounds  
 0 <= on\_t1\_th4 <= 1  
 0 <= on\_t2\_th4 <= 1  
 0 <= starting\_t0\_th4 <= 1  
 0 <= off\_t1\_th4 <= 1  
 0 <= off\_t2\_th4 <= 1  
 0 <= stopping\_t0\_th4 <= 1  
Binaries  
 on\_t1\_th4 on\_t2\_th4 starting\_t0\_th4 off\_t1\_th4 off\_t2\_th4  
 stopping\_t0\_th4

# Compilation

Pour Java, à la compilation, il suffit d'ajouter le fichier JAR de CPLEX. Par exemple, sous Windows, en notant %CPLEX% le dossier d'installation de CPLEX (par exemple, C:\Program Files (x86)\IBM\ILOG\CPLEX\_Studio\_126) :

> javac… -classpath "%CPLEX%\cplex\lib"

Lors de l'exécution, il faudra, en plus, spécifier l'emplacement de la bibliothèque partagée native CPLEX. Puisque l'interaction se fait par JNI, la plateforme de CPLEX utilisée doit parfaitement correspondre avec celle de la machine virtuelle Java (si la machine virtuelle est en 32 bits, la version 32 bits de CPLEX devra être utilisée). Par exemple :

> java… -classpath "%CPLEX%\cplex\lib" -Djava.library.path="%CPLEX%\cplex\bin\x86\_win32"

En C++, la situation est similaire : il faut indiquer au compilateur le chemin des en-têtes (par exemple, %CPLEX%\cplex\include et %CPLEX%\concert\include) et des fichiers d'import (par exemple, %CPLEX%\cplex\lib\x86\_windows\_vs2010\stat\_mda) de CPLEX, en plus de lui indiquer de lier avec une série de fichiers d'import (par exemple, pour Visual Studio sous Windows, ilocplex.lib, concert.lib et cplex*125*.lib – il faut veiller à remplacer les chiffres par la version de CPLEX utilisée). Les bibliothèques partagées devront être disponibles à l'application compilée (par exemple, sous Windows, il faut le fichier cplex*125*.dll, qui peut se situer sous %CPLEX%\cplex\bin\x86\_win32).

# Exemple complet

Soit un problème de sac à dos. Le sac à dos sera un objet (ce qui permettra de généraliser le modèle à de l'assignation, par exemple, en imposant que chaque objet est au plus dans un sac à dos), il n'aura comme attribut que sa capacité.

**Program listing. Language: c++.**

class KnapSack {  
private:   
 const int capacity;   
  
public:  
 KnapSack(int c) : capacity(c) {}  
};

**Program listing. Language: java.**

public class KnapSack {  
 private final int capacity;  
  
 public KnapSack(int c) {  
 capacity = c;  
 }  
}

Chaque objet sera également un objet, avec son poids, sa valeur et un identifiant.

**Program listing. Language: c++.**

class Object {  
private:  
 const int id, weight, value;  
  
public:  
 Object(int i, int w, int v) : id(i), weight(w), value(v) {}  
 int getId() { return id; }  
 int getWeight() { return weight; }  
 int getValue() { return value; }  
};

**Program listing. Language: java.**

public class Object {  
 private final int id, weight, value;  
  
 public Object(int i, int w, int v) {  
 id = i; weight = w; value = v;  
 }  
  
 public int getId() { return id; }  
 public int getWeight() { return weight; }  
 public int getValue() { return value; }md  
}

Il faut alors créer une variable dans le modèle mathématique par objet. Il est commode d'effectuer cette opération dans une méthode et de stocker la valeur dans l'objet.

**Program listing. Language: c++.**

#include <ilcplex/ilocplex.h>  
  
class Object {  
private:  
 IloNumVar\* x;  
  
public:  
 IloNumVar& getX() { return \*x; }  
  
 void createVariables(IloEnv& env) {  
 x = new IloNumVar(env, 0, 1, IloNumVar::Bool, "x" + id);  
 }  
};

**Program listing. Language: java.**

import ilog.concert.IloException;  
import ilog.concert.IloIntVar;  
import ilog.cplex.IloCplex;  
  
public class Object {  
 private IloIntVar x;  
 public IloIntVar getX() { return x; }  
  
 public void createVariables(IloCplex solver) throws IloException {  
 x = solver.boolVar("x" + id);  
 }  
}

Côté sac à dos, il faudra imposer que les objets pris ne dépassent pas la capacité. Pour ce faire, une méthode imposera cette contrainte, elle prendra en argument tous les objets.

**Program listing. Language: c++.**

#include <ilcplex/ilocplex.h>  
#include <vector>  
  
class KnapSack {  
public:  
 void addConstraints(IloModel& model, vector<Object\*> objects) {  
 IloIntVarArray xs;  
 IloIntArray weights;  
 for (auto object : objects) {  
 xs.add(object->getX());  
 weights.add(object->getWeight());  
 }  
  
 model.add(IloScalProd(weights, xs) <= capacity);  
 }  
};

**Program listing. Language: java.**

import ilog.concert.IloException;  
import ilog.concert.IloIntVar;  
import ilog.cplex.IloCplex;  
  
public class KnapSack {  
 public void addConstraints(IloCplex solver, Object[] objects) throws IloException {  
 IloIntVar[] xs = new IloIntVar[objects.length];  
 int[] weights = new int[objects.length];  
 for(int i = 0; i < objects.length; ++i) {  
 xs[i] = objects[i].getX();  
 weights[i] = objects[i].getWeight();  
 }  
  
 solver.addLe(solver.scalProd(weights, xs), capacity);  
 }  
}

La fonction principale se chargera de créer ces objets, d'instancier le solveur, d'appeler les méthodes pour créer un modèle, puis de le résoudre et d'afficher la solution.

**Program listing. Language: c++.**

#include <ilcplex/ilocplex.h>  
#include <ilconcert/iloexpression.h>  
#include <ilconcert/ilolinear.h>  
#include <vector>  
  
int main(int argc, char\* argv[]) {  
 // Create data.  
 vector<Object\*> objects;  
 objects[0] = new Object(1, 12, 5);  
 objects[1] = new Object(2, 25, 9);  
 objects[2] = new Object(3, 10, 3);  
  
 KnapSack sack(30);  
  
 // Initialise solver.  
 IloEnv env;  
 IloModel model(env);  
  
  
 // Fill the model.  
 for (auto o : objects) {  
 o->createVariables(env);  
 }  
 sack.addConstraints(model, objects);  
  
 IloNumVarArray xs;  
 IloIntArray values;  
 for (auto object : objects) {  
 xs.add(object->getX());  
 values.add(object->getValue());  
 }  
 model.add(IloMaximize(env, IloScalProd(xs, values)));  
  
 // Solve it.  
 IloCplex solver(model);  
 solver.solve();  
 solver.exportModel("model.lp");  
 solver.setParam(IloCplex::TiLim, 30);  
  
 // Give the solver one feasible solution (useful in large MIP models).  
 IloNumArray start(env, 3, 1., 0., 1.);   
 solver.addMIPStart(xs, start, IloCplex::MIPStartAuto, "initial solution");  
  
 // Show the solution.  
 for (auto object : objects) {  
 cout << "Object " << object->getId() << ": " << ((solver.getValue(object->getX()) > 0.75) ? "taken" : "left");  
 }  
 cout << "Value: " << solver.getObjValue();  
}

**Program listing. Language: java.**

import ilog.concert.IloException;  
import ilog.concert.IloIntVar;  
import ilog.cplex.IloCplex;  
  
public class Main {  
 public static void main(String[] args) throws IloException {  
 // Create data.  
 Object[] objects = new Object[3];  
 objects[0] = new Object(1, 12, 5);  
 objects[1] = new Object(2, 25, 9);  
 objects[2] = new Object(3, 10, 3);  
  
 KnapSack sack = new KnapSack(30);  
  
 // Initialise solver.  
 IloCplex solver = new IloCplex();  
  
 // Fill the model.  
 for(Object o : objects) {  
 o.createVariables(solver);  
 }  
 sack.addConstraints(solver, objects);  
  
 IloIntVar[] xs = new IloIntVar[objects.length];  
 int[] values = new int[objects.length];  
 for(int i = 0; i < objects.length; ++i) {  
 xs[i] = objects[i].getX();  
 values[i] = objects[i].getValue();  
 }  
 solver.addMaximize(solver.scalProd(xs, values));  
  
 // Give the solver one feasible solution (useful in large MIP models).  
 solver.addMIPStart(xs, new double[]{1., 0., 1.}, "initial solution");  
  
 // Solve it.  
 solver.exportModel("model.lp");  
 solver.setParam(IloCplex.DoubleParam.TiLim, 30);  
 solver.solve();  
  
 // Show the solution.  
 for (Object object : objects) {  
 String taken = (solver.getValue(object.getX()) > 0.75) ? "taken" : "left";  
 System.out.println("Object " + object.getId() + ": " + taken);  
 }  
 System.out.println("Value: " + solver.getObjValue());  
 }  
}

La méthode addMIPStart() n'est utile que pour la résolution de problèmes impliquant des nombres entiers (d'où son nom). Elle fournit une solution initiale, faisable ou non, complète ou non, qui peut aider le solveur à démarrer, à trouver une première solution faisable, afin d'accélérer le prétraitement et l'algorithme de séparation et évaluation. Le solveur indiquera, lors de son exécution, quelles solutions initiales ont fourni des solutions et ont donné une première indication sur la valeur de l'objectif.

Ce jeu de données peut être déterminé par une connaissance du problème (pour une situation à améliorer, l'actuel fonctionne, il donne donc une solution faisable : pour déterminer un meilleur horaire des trains, l'horaire actuel est déjà connu et fonctionne ; lors de l'ajout de nouveaux trains, la solution optimale pour les trains précédents est connue).