Paramétrage du *Branch* & *Cut* pour la résolution de problèmes en nombres entiers (ou mixte) dans CPLEX 12.6

Hélène Toussaint, 2018

But : Ce document a pour but de compléter le tutoriel [1] de présentation générale de la librairie *Concert Technology* pour l'utilisation de CPLEX dans un code C++. On se focalise ici les principaux paramètres permettant au développeur d'agir sur la résolution d'un programme linéaire en nombres entiers ou mixte (*Mixed Integer Programs,* MIP). Ce document s'appuie en grande partie sur la documentation en ligne d'IBM ([2]).

Prérequis :

1. Savoir utiliser la *Concert Technology* C++ pour résoudre un programme linéaire (voir [1] ou [3])
2. Etre familier des techniques de *Branch & Bound* et *Branch & Cut* dans le cadre de la résolution de problèmes linéaires en nombres entiers ou mixtes (*Mixed Integer Programs,* MIP, voir [4]).

Remarque : l'utilisation des fonctions *callbacks* n'est pas abordée dans ce document.

Sommaire

Dans la suite du document on suppose définie la variable `cplex` du type `IloCplex`.

1 SCHEMA GENERAL DE RESOLUTION D'UN PROBLEME EN NOMBRES ENTIERS OU MIXTE (MIP - *MIXED INTEGER PROGRAM*)

La résolution d'un MIP dans CPLEX repose sur la construction d'un arbre de recherche, chaque nœud de l'arbre représentant un sous-problème à résoudre. CPLEX possède deux algorithmes distincts :

- un *Branch & Cut* classique (B&C)
- une recherche dynamique (*dynamic search),* algorithme propre à Cplex, intégré à la version 12, et non documenté (à ma connaissance). Les seules informations fournies dans la documentation d'IBM ([6]) à son sujet sont qu'il possède les mêmes composantes principales que le *Branch & Cut* (résolution de la relaxation, ajout de coupes et heuristiques, séparation). Cependant il n'est pas possible de contrôler cette recherche dynamique grâce aux fonctions *callbacks*.

L'utilisateur peut imposer à CPLEX le type de recherche qu'il souhaite grâce au paramètre `IloCplex::MIPSearch`. On impose un B&C classique avec la valeur `IloCplex::Traditional` et une recherche dynamique avec `IloCplex::Dynamic`. On fixe la valeur du paramètre grâce à la méthode `setParam` :

```
cplex.setParam(IloCplex::MIPSearch, IloCplex::Traditional);
```

Le schéma général du *Branch & Cut* est représenté Figure 1. Chaque étape du *Branch & Cut* (modélisée par un rectangle) sera détaillée dans la partie suivante.

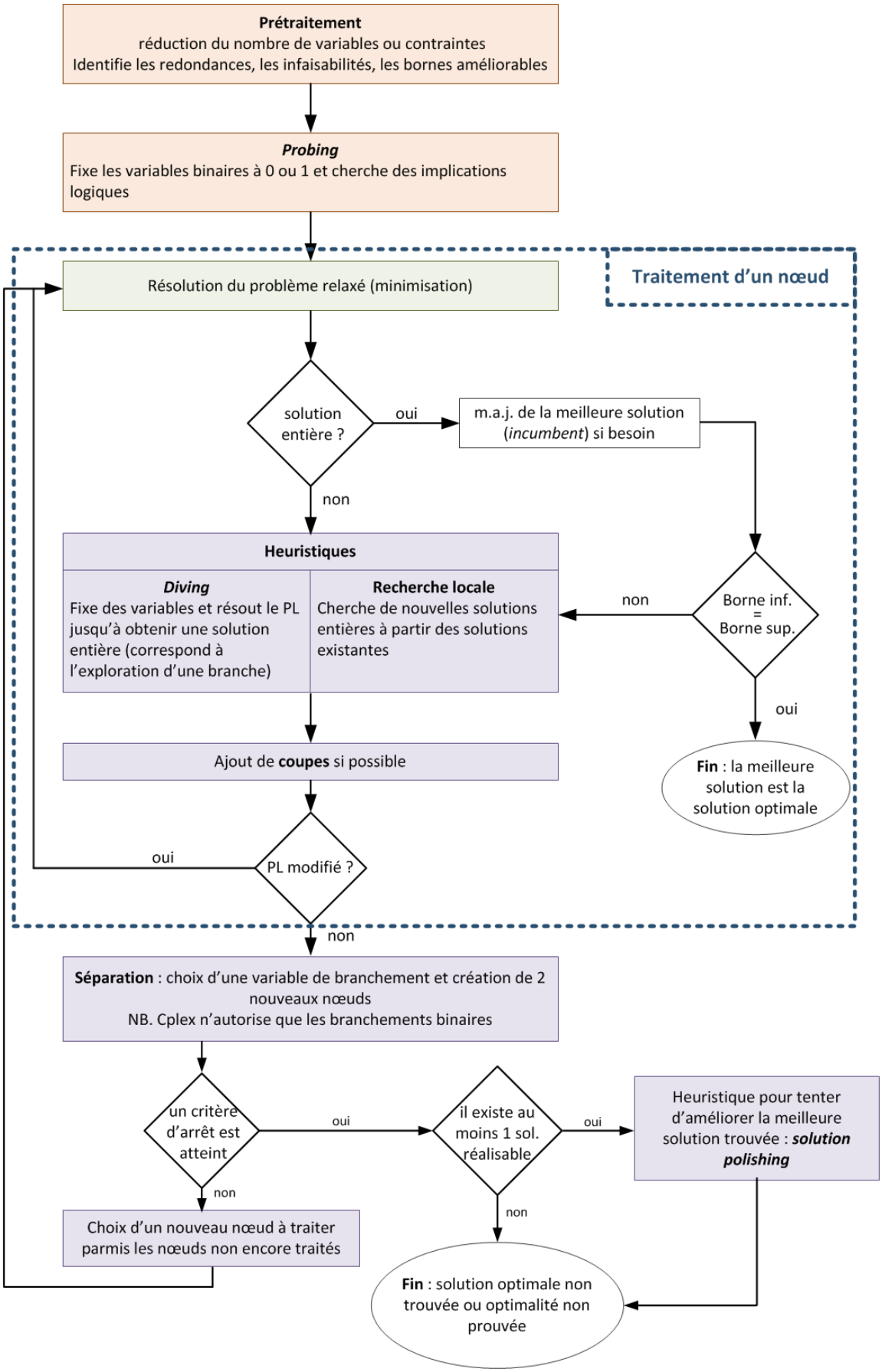

Figure 1. Schéma général du *Branch & Cut* dans CPLEX dans le cas d'un problème de minimisation

2 PARAMETRAGE DU BRANCH & CUT

Dans cette partie on détaille les différentes étapes du *Branch & Cut* et on donne les principaux paramètres permettant d'agir sur chacune de ces étapes.

2.1 LES PARAMETRES DANS CPLEX 12.6 : NOMS, ACCES ET MODIFICATION

Une documentation complète concernant l'ensemble des paramètres existants dans CPLEX est disponible sur le site d'IBM (voir [14]).

2.1.1 NOMS DES PARAMETRES DEPUIS LA VERSION 12.6

A partir de la version 12.6 de CPLEX les paramètres sont définis par des noms longs dans une hiérarchie de classes qui dépend de leur domaine d'applicabilité.
Par exemple le paramètre `IloCplex::Param::MIP::Tolerances::UpperCutoff` permet de fournir une valeur de *cutoff*[1] pour un problème en nombre entier ou mixte (MIP). Ces classes sont définies dans le fichier *include* **`ilcplex\iloparam.h`**.

Les noms courts des paramètres (qui existaient avant la version 12.6) sont toujours disponibles et peuvent être utilisés. Ils doivent être simplement précédés du nom de classe `IloCplex` suivi de l'opérateur de portée `::`. Par exemple la version courte du paramètre précédent est `IloCplex::CutUp`. Ces noms courts sont définis dans des **`enum`** dans le fichier *include* **`ilcplex\ilocplexi.h`**.

Dans la suite du document on utilisera les noms courts des paramètres pour des raisons de lisibilité (voir [14] pour trouver le nom long à partir du nom court).

2.1.2 GESTION DES PARAMETRES

Il existe cinq types de paramètres (booléens, entiers, entiers longs, réels, chaine de caractères). On affecte toujours une valeur à un paramètre en utilisant la fonction **`setParam(<nom du paramètre>, <valeur à affecter>)`** de la classe `IloCplex` quel que soit le type du paramètre considéré (cette fonction étant surchargée pour tous les types possibles).

Les 3 fonctions suivantes permettent de manipuler les paramètres et sont définies pour tous les types possibles :

- `getParam (<nom du paramètre>)` : retourne la valeur actuelle du paramètre considéré
- `getDefault (<nom du paramètre>)` : retourne la valeur par défaut du paramètre considéré
- `setDefaults (<nom du paramètre>)` : attribut sa valeur par défaut au paramètre considéré

[1] dans le cas d'un problème de minimisation, une valeur de *cutoff* C permet d'écarter toutes les solutions dont la valeur objective est supérieure strictement C

Les deux fonctions suivantes peuvent être utilisées avec des paramètres numériques uniquement :

- `getMin (<nom du paramètre>)` : retourne la valeur minimale autorisée pour le paramètre en entrée
- `getMax (<nom du paramètre>)` : retourne la valeur minimale autorisée pour le paramètre en entrée

La fonction :

```
inline void IloCplex::writeParam(const char* name) const
```

permet d'écrire dans un fichier la valeur des paramètres qui ont été modifiés par l'utilisateur (et donc fixés à une valeur différente de celle par défaut). Le nom du fichier est passé en paramètre de la fonction, l'extension habituelle (pour CPLEX) pour les fichiers de paramètre est "**`.prm`**".

On peut relire un fichier de paramètres avec la fonction :

```
inline void IloCplex::readParam(const char* name) const
```

2.2 LE PRETRAITEMENT (PREPROCESSING)

Le prétraitement ([16]) a pour but de réduire le modèle en termes de nombre de contraintes et de variables et d'obtenir une meilleure valeur de relaxation afin d'avoir un B&C plus performant. Pour cela CPLEX dispose de deux outils :

- Le présolveur (*presolver*) essaie de réduire la taille d'un problème en effectuant des inférences sur la nature d'une solution du problème optimale ;
- L'agrégateur (*agregator*) essaie d'éliminer des variables ou contraintes par substitution.

Lors du prétraitement plusieurs mécanismes entrent en jeu :

- L'**agrégation** utilise des substitutions pour réduire le nombre de lignes et de colonnes ;
- Le **renforcement** améliore la relaxation en remplaçant une ligne du modèle par une autre de sorte qu'un vecteur d'entiers soit admissible dans la nouvelle ligne si et seulement si ce même vecteur d'entiers était admissible dans la ligne d'origine ;
- La **réduction des coefficients** réduit le nombre de sommets non entiers du polyèdre associé à la relaxation fractionnaire du PLNE considéré ;
- La **répétition du prétraitement** permet de lancer une nouvelle fois le prétraitement à la racine après avoir enrichit le modèle de coupes d'intégrité (voir section 2.6).

Le Tableau 1 donne les paramètres permettant d'agir sur le prétraitement.

Nom court du paramètre	**fonctions**	**Valeurs possibles**
`IloCplex::AggInd`	Nombre de fois où l'agrégateur est appelé	$n \in \mathbb{N}$ -1: CPLEX choisit (défaut)
`IloCplex::PreInd`	Active / désactive la *prérésolution*	0 : désactivé 1 : activé (défaut)

`IloCplex::BndStrenInd`	Active / désactive le renforcement	-1 : CPLEX choisit (défaut) 0 : désactivé 1 : activé
`IloCplex::CoeRedInd`	Agressivité de la réduction des coefficients	-1 : CPLEX choisit (défaut) 0 : pas de réduction 1 à 3 : réduction de plus en plus agressive
`IloCplex::RelaxPreInd`	Active / désactive la *prérésolution* de la relaxation à la racine	-1 : CPLEX choisit (défaut) 0 : désactivé 1 : activé
`IloCplex::Reduce`	Active / désactive les réductions primales et : ou duales	0 : pas de réduction 1 : réductions primales 2 : réductions duales 3 : réductions primales et duales (défaut)
`IloCplex::PrePass`	Nombre de *prérésolutions* à effectuer	$n \in \mathbb{N}$ -1: CPLEX choisit (défaut)
`IloCplex::RepeatPresolve`	Active / désactive la répétition du prétraitement	-1 : CPLEX choisit (défaut) 0 : désactivé 1 à 3 : répétition d'un prétraitement de plus en plus agressif

Tableau 1 : Paramètres liés au prétraitement

Remarque : CPLEX effectue par défaut un prétraitement sauf si on lui fournit une base initiale pour résoudre la relaxation à la racine. Dans ce cas CPLEX utilise cette base de départ sans prétraitement, on perd alors les bénéfices apportés par le prétraitement qui sont parfois meilleurs que ceux apportés par l'ajout d'une base initiale.

2.3 LE *PROBING*

Le *probing* est une technique utilisée dans les PLNE comportant des variables binaires pour détecter des implications logiques et ainsi réduire le nombre de variables. Il est effectué au terme du prétraitement et peut être relativement coûteux en temps de calcul. L'utilisateur peut choisir l'intensité du *probing* grâce au paramètre **`IloCplex::Probe`**. Il prend une valeur entière qui dépend de l'intensité du probing souhaité par l'utilisateur :

- -1 : pas de probing ;
- 0 (défaut) : CPLEX choisit ;
- 1 : probing modéré ;
- 2 : probing agressif ;
- 3 : probing très agressif.

2.4 RESOLUTION DE LA RELAXATION

Après le prétraitement et le *probing* la relaxation fractionnaire du problème est résolue. CPLEX dispose pour cela de plusieurs algorithmes. Le paramètre qui permet de fixer l'algorithme à utiliser est **`IloCplex::RootAlg`** pour la racine et **`IloCplex::NodeAlg`** pour les autres nœuds. Ils peuvent prendre les valeurs :

- `IloCplex::AutoAlg` (0, défaut) : CPLEX choisit ;

- `IloCplex::Primal` (1) : simplexe primal ;
- `IloCplex::Dual` (2) : simplexe dual ;
- `IloCplex::Network` (3) : simplexe réseau ;
- `IloCplex::Barrier` (4) : algorithme barrière ;
- `IloCplex::Sifting` (5) : criblage ;
- `IloCplex::Concurrent` (6) : concurrent (Dual, Algorithme barrière et Primal en mode opportuniste ; Dual et Algorithme barrière en mode déterministe) - **Non compatible** avec `IloCplex::NodeAlg`.

Remarque : dans la fonction **`setParam()`** on peut utiliser indifféremment le nom de l'algorithme ou le numéro associé, par exemple :

```
cplex.setParam(IloCplex::NodeAlg, IloCplex::Dual);
```

équivaut à

```
cplex.setParam(IloCplex::NodeAlg, 2);
```

mais il est préférable d'utiliser le nom car cela donne plus de lisibilité au programme.

2.5 RECHERCHE DE SOLUTIONS HEURISTIQUES

A la racine, et régulièrement pendant le B&C, CPLEX recherche des solutions heuristiques à partir de la solution fractionnaire du nœud courant. On peut définir la fréquence d'appel des heuristiques grâce au paramètre **`IloCplex::HeurFreq`** qui peut prendre les valeurs :

- -1 : pas d'heuristique ;
- 0 (défaut) : CPLEX choisit ;
- $n > 0$: tous les n nœuds.

CPLEX utilise différentes heuristiques à différents niveaux de l'exploration arborescente :

- La **méthode RINS** (*Relaxation Induced Neighborhood Search*) explore un voisinage de la meilleure solution courante qui dépend en partie de la solution fractionnaire au nœud courant (voir [7]). Cette heuristique utilise un B&C tronqué.

- l'heuristique ***solution polishing*** tente d'améliorer une solution, elle a donc besoin d'une solution réalisable pour fonctionner. Elle consiste également en un B&C mais focalisé sur la recherche de "bonnes" solutions et ne peut donc pas garantir l'optimalité. Comme elle est assez couteuse en temps de calcul, elle est appelée à la fin du B&C classique pour essayer d'améliorer la meilleure solution courante. Cette heuristique est intéressante lorsque le B&C classique peine à trouver des solutions de bonne qualité. L'article [8] décrit son fonctionnement.

- l'heuristique ***feasibility pump*** a pour but de trouver une solution réalisable notamment pour les problèmes mixtes "difficiles". Elle fonctionne en résolvant une séquence de problèmes linéaires issus de la relaxation du PLNE (voir [9], [10] et [11]).

2.6 AJOUT DE COUPES

Afin de réduire le gap entre le coût de la solution optimale du MIP et celui de sa relaxation fractionnaire, CPLEX ajoute des coupes (i.e. des contraintes qui permettent d'éliminer des solutions fractionnaires). Cela permet généralement de réduire le nombre de séparations nécessaires et ainsi de gagner du temps dans la résolution.

CPLEX ajoute au PL des coupes globales (c'est-à-dire valides quel que soit le nœud considéré) et des coupes locales (valides uniquement pour un nœud et ses descendants).

CPLEX 12.6 offre 12 types de coupes différentes (pour plus de détails voir [5]) :

- Coupes de clique ;
- Coupes de couverture ;
- Coupes disjonctives ;
- Coupes de couverture de flots ;
- Coupes de chemin de flots ;
- Coupes fractionnaires de Gomory ;
- Coupes de couverture de borne supérieure généralisée ;
- Coupes de borne implicite ;
- Coupes Lift-and-Project PMNE ;
- Coupes MIR (Mixed Integer Rounding) ;
- Coupes de flot multicommodité (MCF, Multi-Commodity Flow) ;
- Coupes de demi-zéro.

La fréquence de recherche de chaque type de coupe peut être paramétrée (-1 : pas de coupes de ce type ; 0 : CPLEX choisit ; entre 1 et 3 : recherche de coupes de plus en plus agressive, 3 n'étant pas disponible pour tous les types de coupe). Par exemple pour les coupes de couverture :

```
cplex.setParam( IloCplex::Covers, 2 );
```

A la fin de l'optimisation on peut, pour chaque classe de coupes, accéder au nombre de coupes générées par CPLEX grâce à la fonction **`IloCplex::getNcuts()`** qui prend en paramètre le nom de la coupe pour laquelle on souhaite avoir l'information. Par exemple :

```
cout << cplex.getNcuts(IloCplex::CutType::CutCover) << endl;
```

2.7 SEPARATION

2.7.1 LES REGLES PREDEFINIES

Pendant la phase de séparation CPLEX choisit une variable à séparer parmi les variables fractionnaires de la solution courante. Il dispose de plusieurs règles *ad hoc* pour déterminer quelle

est la variable à choisir. L'utilisateur peut imposer une des règles prédéfinies grâce au paramètre `IloCplex::VarSel`. Les valeurs possibles sont :

- `IloCplex::MinInfeas (-1)` : choix de la variable la moins fractionnaire (permet en général d'accéder plus rapidement à des solutions entières mais pas nécessairement de bonne qualité) ;
- `IloCplex::DefaultVarSel (0)` : CPLEX décide de la décision à prendre en chaque nœud en fonction du problème et de l'avancement dans l'arbre de recherche ;

- `IloCplex::MaxInfeas (1)` : choix de la variable la plus fractionnaire. Permet en général un plus grand changement de la solution courante dans le but d'élaguer l'arbre un maximum pour atteindre la solution optimale plus rapidement.

- `IloCplex::Pseudo (2)` : utilise les "*pseudo costs*" ou "pseudo coûts". Le pseudo coût d'une variable donne une estimation du gain apporté par un branchement sur cette variable. Ils sont calculés à la racine par le procédé de *strong branching.*

- `IloCplex::Strong (3)` : utilise le "*strong branching*" i.e. pour chaque variable CPLEX résout un PL dans lequel on fixe cette variable à la valeur de branchement envisagée afin de déterminer quel branchement est le plus prometteur du point de vue de Cplex . Cette méthode requiert un temps de calcul important. En général pour gagner du temps on choisit seulement un sous-ensemble de variables de manière heuristique et les PL sont tronqués (i.e. arrêté après un certains nombres d'itérations du simplexe). Je ne sais pas exactement quel est la stratégie utilisée par CPLEX.
  NB. Lorsque le procédé teste toutes les variables cela s'appelle "*Full strong branching"*.

- `IloCplex::PseudoReduced (4) :` utilise les "*pseudo reduced costs*" ou pseudos coûts réduits. Il s'agit du même mécanisme que précédemment mais les pseudo coûts réduits sont calculés plus rapidement que les pseudo coûts (à ma connaissance le calcul de ces coûts n'est pas documenté).

Exemple d'utilisation :

```
cplex.setParam(IloCplex::VarSel, IloCplex::MaxInfeas );
```

2.7.2 LES PRIORITES DES VARIABLES

L'utilisateur peut imposer des priorités aux variables entières. Au moment de la séparation CPLEX choisira alors la variable la plus prioritaire (priorité la plus élevée) pour le branchement. Par défaut les variables ont toutes une priorité de 0 et CPLEX ne peut effectuer de séparation que sur des variables qui ont une valeur fractionnaire dans la solution de la relaxation courante et qui n'ont pas été supprimée par le prétraitement.

L'affectation d'une priorité à une variable s'effectue via les méthodes :

```
public void setPriority(IloIntVar var, IloNum pri)
public void setPriority(IloNumVar var, IloNum pri)
```

de la classe `IloCplex`.

2.8 CHOIX DU NOEUD SUIVANT

L'utilisateur peut choisir la stratégie à utiliser pour parcourir les nœuds dans l'arbre de recherche à l'aide du paramètre **`IloCplex::NodeSel`**. Les valeurs possibles sont :

- `IloCplex::DFS` (0) : parcourt en profondeur ;
- `IloCplex::BestBound` (1) : choix du nœud avec la meilleure valeur de relaxation (défaut) ;
- `IloCplex::BestEst` (2) : CPLEX estime pour chaque nœud la valeur de la fonction objectif pour une solution entière et choisit celui qui donne la meilleure valeur (la manière dont Cplex calcule l'estimation n'est pas documentée par IBM) ;
- `IloCplex::BestEstAlt` (3) : idem `IloCplex::BestEst` mais avec une estimation calculée différemment.

2.9 CRITERE D'ARRET

L'optimisation s'arrête dans diverses circonstances :

1. la solution optimale a été trouvée. L'optimalité est déclarée si l'écart (en valeur absolue) entre le coût de la meilleure solution et la borne (borne inférieure en cas de minimisation, supérieure en cas de maximisation) est inférieur aux valeurs de *gap* définis par :
   - `IloCplex::EpGap` : écart relatif entre la meilleure solution entière trouvée et la meilleure borne (par défaut 0.0001 soit 0.01 %) ;
   - `IloCplex::EpAGap` : écart absolu entre la meilleure solution entière trouvée et la meilleure borne (par défaut 10^{-6}).

2. une limite fixée par l'utilisateur a été atteinte. L'utilisateur peut fixer une des limites suivantes (ces limites valent toutes $+\infty$ par défaut) :
   - `IloCplex::TiLim` : le temps total (en secondes) accordé à la résolution ;
   - `IloCplex::DetTiLim` : le temps total (en tics d'horloge) accordé à la résolution (ne prend pas en compte le temps passé dans les fonctions *callbacks*) ;
   - `IloCplex::NodeLim` : le nombre de nœuds résolus. Si ce paramètre vaut 0 alors seul le traitement à la racine est effectué, s'il vaut 1 l'étape de séparation à la racine est effectuée mais les nœuds fils ne sont pas résolus ;
   - `IloCplex::TreLim` : la taille maximale (en Mo) de l'arbre ;
   - `IloCplex::IntSolLim` : le nombre de solutions entières trouvées durant la résolution.

3. un échec est survenu en cours de résolution (problème mémoire ou échec de résolution d'un sous problème).

Pour plus de détails sur les critères d'arrêt voir [17].

2.10 COMPROMIS ENTRE SOLUTIONS REALISABLES ET PREUVE D'OPTIMALITE

Le paramétrage du *Branch & Cut* permet d'orienter CPLEX dans sa recherche arborescente. Il existe également un paramètre général pour choisir la direction dans laquelle se concentrent les efforts de CPLEX : **`IloCplex::MIPEmphasis.`** Ce paramètre permet de décider du compromis entre la recherche de solutions réalisables et la preuve d'optimalité. Lorsque CPLEX résout un MIP "difficile", et qu'il peine à trouver des solutions réalisables, il peut être intéressant de l'orienter vers la recherche de solutions réalisables.

Le paramètre **`IloCplex::MIPEmphasis`** peut prendre les valeurs suivantes :

- `IloCplex::MIPEmphasisBalanced (0)` : compromis entre optimalité et réalisabilité (défaut) ;
- `IloCplex::MIPEmphasisFeasibility (1)` : CPLEX tente de générer plus de solutions réalisables (augmente les temps de calcul) ;
- `IloCplex::MIPEmphasisOptimality (2)` : CPLEX réduit la recherche de solutions réalisables dans le but de fournir plus rapidement la preuve d'optimalité (pas toujours avantageux) ;
- `IloCplex::MIPEmphasisBestBound (3)` : CPLEX met encore davantage l'accent sur l'optimalité au détriment de la recherche de solutions réalisables ;
- `IloCplex::MIPEmphasisHiddenFeas (4)` : CPLEX tente de trouver des solutions réalisables de très bonne qualité (à utiliser si `IloCplex::MIPEmphasisFeasibility` ne suffit pas pour trouver des solutions de qualité).

2.11 CHOIX DES PARAMETRES POUR UN PROBLEME "DIFFICILE" ET *TUNING*

2.11.1 TROUVER LES BONS PARAMETRES "A LA MAIN"

Le site d'IBM dédié au support ([12]), donne une méthodologie pour tenter d'améliorer la résolution des MIP "difficiles" en paramétrant le *Branch & Cut* de manière astucieuse. Je donne ici les grandes lignes :

1. Utiliser, si possible, la dernière version de CPLEX (une version de CPLEX récente avec ses paramètres par défaut peut donner de meilleurs résultats qu'une ancienne version astucieusement paramétrée) ;
2. Identifier d'où vient le problème en étudiant le *node log* (voir section 3) ;
3. Si le problème vient de la résolution de la relaxation, essayer les différents algorithmes (voir section 2.4) ;
4. Si le problème contient des variables binaires, essayer de régler le *probing* sur sa forme agressive (voir section 2.3) ;
5. Si on veut simplement une solution de bonne qualité mais pas forcément optimale, fixer le paramètre **`IloCplex::MIPEmphasis`** à 1 ou augmenter le gap **`IloCplex::EpGap`** (par exemple le fixer à 0.05 pour que CPLEX s'arrête dès qu'il est à moins de 5% de la solution optimale) ;
6. heuristique et mip start

7. essayer d'augmenter les coupes générées (voir section 2.6) afin de renforcer la relaxation
8. si CPLEX passe beaucoup de temps à la racine essayer de fixer le paramètre `IloCplex::VarSel` à 4 (voir section 2.7.1). Si au contraire CPLEX passe peu de temps à la racine essayer de fixer ce paramètre à 3 (*strong branching*).
9. utiliser la connaissance du modèle pour adapter la séparation ou choisir les variables sur lesquelles il est intéressant de brancher en priorité. Par exemple, dans le cas d'un problème indexé sur le temps il peut être judicieux de brancher en priorité sur les variables en début d'horizon.

Si rien ne fonctionne, essayer de reformuler le modèle.

2.11.2 UTILISATION DE L'OUTIL DE "TUNING"

CPLEX dispose d'une fonction tuneParam() qui recherche le meilleur jeu de paramètres pour résoudre un modèle donné. Il suffit d'appeler cette méthode dans le code à la place de la méthode solve() puis d'enregistrer les paramètres qu'elle fournit à l'aide de la fonction writeParam(). L'exemple suivant illustre comment faire :

```
    //1. cplex teste differents jeux de parametres en appelant la methode tuneParam()
    IloInt tunestat = cplex.tuneParam();

    // 2. affichage de l'état du tuning :
    if      ( tunestat == IloCplex::TuningComplete)
        cout << "Tuning complete." << endl;
    else if ( tunestat == IloCplex::TuningAbort)
        cout << "Tuning abort." << endl;
    else if ( tunestat == IloCplex::TuningTimeLim)
        cout << "Tuning time limit." << endl;
    else
        cout << "Tuning status unknown." << endl;

    //3.ecrire dans un fichier le meilleur jeu de paramètres
    cplex.writeParam("bestParam.prm");
```

Exemple d'utilisation de **`tuneParam()`** (extrait du fichier d'exemple ilolpex2.cpp fourni avec CPLEX)

Pour avoir un *tuning* encore plus robuste on peut fixer le paramètre `IloCplex::TuningRepeat` à une valeur strictement supérieure à 1, CPLEX répète alors le *tuning* on opérant des permutations dans les lignes et colonnes :

```
cplex.setParam(IloCplex::TuningRepeat, 2);
```

J'ai testé le *tuning* sur quelques instances proposées comme benchmark sur le site de (voir http://miplib.zib.de/miplib2010.php)

3 LE JOURNAL DES NŒUDS (*NODE LOG FILE*)

Au fur et à mesure de la résolution, CPLEX fournit des informations sur sa progression dans l'arbre de recherche. La quantité d'information fournie par CPLEX peut être ajustée à l'aide du paramètre

`IloCplex::MIPDisplay`. Les valeurs possibles vont de 0 (pas d'affichage) à 5 (affichage le plus exhaustif). Par défaut ce paramètre est fixé à 2.

CPLEX affiche donc des informations tous les N nœuds à raison d'une ligne par nœud. La valeur N est appelée *intervalle d'affichage* et peut être fixée à l'aide du paramètre `IloCplex::MIPInterval`. Par défaut ce paramètre est fixé à 100.

La Figure 2 donne un d'exemple d'affichage avec `IloCplex::MIPDisplay = 2` et `IloCplex::MIPInterval = 10`. Il y a 8 colonnes qui ont la signification suivante :

- la colonne ***Node*** donne le numéro du nœud, une étoile (*) à gauche de ce numéro signifie qu'une nouvelle solution entière a été trouvée, un signe plus (+) à droite indique que la nouvelle solution entière a été trouvée par une heuristique ;
- la colonne ***Nodes Left*** donne le nombre de nœuds déjà crées qu'il reste à explorer ;
- la colonne ***Objective*** donne la valeur de la fonction objectif pour le problème relaxé associé au nœud, ou la raison pour laquelle le nœud a été *écarté*[2] (*fathomed*). Si c'est une heuristique qui a donné la solution alors la colonne est laissée vide ;
- la colonne ***IInf*** donne le nombre de variables entières avec une valeur fractionnaire dans la solution courante ;
- la colonne ***Best Integer*** donne la valeur de la meilleure solution entière trouvée jusqu'à présent ;
- la colonne ***Cuts / Best bound*** donne la meilleure borne courante (borne inférieure dans le cas d'une minimisation, supérieure dans le cas d'une maximisation). Si le mot "**Cuts**" apparaît dans cette colonne, cela signifie que différentes coupes ont été générées (et du un type particulier de coupe est précisé alors seule
- la colonne ***ItCnt*** donne le nombre d'itérations cumulées de l'algorithme qui résout les relaxations ;
- la dernière colonne donne le gap entre la meilleure solution entière et la *best bound*.

[2] un nœud est *écarté* si le sous-problème associé au nœud est infaisable, ou si la valeur de relaxation est plus mauvaise que la valeur de *cutoff* ou si la résolution de la relaxation fournit une solution entière.

```
Tried aggregator 1 time.
Presolve time =    0.00 sec. (0.00 ticks)
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: none, using 1 thread.
Root relaxation solution time = 0.00 sec (0.00 ticks)

         Nodes                                         Cuts/
    Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

*     0+    0                           0.0000     3261.8212        8     ---
*     0+    0                        3148.0000     3261.8212        8    3.62%
      0     0     3254.5370     7     3148.0000       Cuts: 5       14    3.38%
      0     0     3246.0185     7     3148.0000       Cuts: 3       24    3.11%
*     0+    0                        3158.0000     3246.0185       24    2.79%
      0     0     3245.3465     9     3158.0000       Cuts: 5       27    2.77%
      0     0     3243.4477     9     3158.0000       Cuts: 5       32    2.71%
      0     0     3242.9809    10     3158.0000     Covers: 3       36    2.69%
      0     0     3242.8397    11     3158.0000     Covers: 1       37    2.69%
      0     0     3242.7428    11     3158.0000       Cuts: 3       39    2.68%
      0     2     3242.7428    11     3158.0000     3242.7428       39    2.68%
     10    11     3199.1875     2     3158.0000     3215.1261       73    1.81%
*    20+   11                        3168.0000     3215.1261       89    1.49%
     20    13     3179.0028     5     3168.0000     3215.1261       89    1.49%
     30    15     3179.9091     3     3168.0000     3197.5227      113    0.93%
*    39     3      integral     0     3186.0000     3197.3990      126    0.36%
     40     3     3193.7500     1     3186.0000     3197.3990      128    0.36%

Cover cuts applied:  9
Zero-half cuts applied:  2
Gomory fractional cuts applied:  1

Solution pool: 5 solutions saved.
MIP-Integer optimal solution:  Objective =   3.1860000000e+03
Solution time =    0.01 sec.  (0.00 ticks)  Iterations = 131   Nodes = 44
```

Figure 2 : exemple de fichier de *log* (journal des nœuds) pour un problème de maximisation (extrait du site d'IBM [13])

4 FOURNIR UNE BORNE OU UNE SOLUTION INITIALE

4.1 VALEUR DE *CUTOFF*

On peut imposer à CPLEX une valeur pour la fonction objectif à ne pas dépasser. Une telle valeur s'appelle valeur de *cutoff.* Pour un problème de minimisation la valeur de *cutoff C* est une borne supérieure, on la fixe avec le paramètre `IloCplex::CutUp`. CPLEX rejette alors les solutions dont le coût est strictement plus grand que *C* ce qui permet de réduire l'arbre de recherche en élaguant tous les nœuds dont la valeur de relaxation est plus grande que *C*.
Pour un problème de maximisation la valeur de *cutoff* est une borne inférieure, on la fixe avec le paramètre `IloCplex::CutLo`.

Remarque : Si CPLEX ne trouve pas de solution respectant la valeur de *cutoff C* (c'est-à-dire une solution de coût inférieure ou égal à *C* dans le cas d'une minimisation) alors il déclare le problème non faisable.

4.2 AJOUTER UN OU PLUSIEURS *MIP STARTS* (SOLUTIONS "DE DEMARRAGE")

On peut fournir à CPLEX une (ou plusieurs) solution initiale quand on résout un PLNE. Cette solution peut être une solution faisable du modèle mais pas nécessairement. Elle peut être infaisable ou

même partielle, dans ce cas elle sert d'indice à CPLEX pour essayer de construire une solution réalisable.

CPLEX traite la / les solutions initiales fournies par l'utilisateur avant de lancer le Branch & Cut. Fournir une solution réalisable avant le début du *Branch & Cut* peut avoir deux effets positifs :

1. la solution fournie une borne qui peut permettre d'élaguer des parties de l'arbre ;
2. certaines heuristiques de CPLEX nécessitent de connaître une solution réalisable, elles peuvent alors être utilisées.

L'utilisateur peut fournir plusieurs *MIP Start* à CPLEX qui traite alors chaque solution qui lui est fournie. La fonction qui permet de définir une solution initiale est `addMIPStart` :

```
IloInt addMIPStart(IloNumVarArray vars=0, IloNumArray values=0,
IloCplex::MIPStartEffort effort=MIPStartAuto, const char * name=0);
```

Chaque appel à cette crée une nouvelle solution (éventuellement incomplète). Cette fonction prend en argument :

- `IloNumVarArray vars` : tableau des variables dont on souhaite fournir une valeur initiale (il n'est pas obligatoire de fournir des valeurs pour toutes les variables du modèle)
- `IloNumArray values` : tableau des valeurs, values[i] donne la valeur initiale de la variable vars[i]
- `IloCplex::MIPStartEffort effort` (facultatif) : paramètre fixant l'effort à fournir par Cplex pour prendre en compte cette nouvelle solution. Les valeurs possibles de cet argument sont :
  - `IloCplex::MIPStartAuto` : CPLEX choisit (défaut)
  - `IloCplex::MIPStartCheckFeas` : CPLEX vérifie la faisabilité de la solution (toutes les variables du modèle doivent alors être définies)
  - `IloCplex::MIPStartSolveFixed` : CPLEX résout le problème fractionnaire associé à la solution (toutes les variables entières doivent alors être définies)
  - `IloCplex::MIPStartSolveMIP` : CPLEX résout un sous problème entier (au moins une variable entière doit avoir été définie)
  - `IloCplex::MIPStartRepair` : CPLEX tente de réparer la solution si elle est infaisable (le nombre de tentative de réparation peut être fixé à l'aide du paramètre `IloCplex::MIPStartRepair`)
- `const char * name` (facultatif) : le nom à donner à cette solution

Remarque 1 : à ma connaissance le paramètre **`IloCplex::MIPStartEffort`** ne peut pas être fixé avec la fonction **`setParam`**, il faut donc le fixer directement dans la liste des arguments de la fonction **`addMIPStart`.**

Remarque 2 : la fonction **`addMIPStart()`** n'accepte que des vecteurs de dimension 1. Si on utilise des tableaux à 2 dimensions (ou plus) pour stocker les variables alors il faut au préalable copier toutes les variables et les valeurs associées dans des vecteurs de dimension 1. L'exemple suivant montre comment s'y prendre quand on a initialement des variables et valeurs stockées dans des tableaux 2D (respectivement x et start).

```
  //1. creation de 2 tableaux 1D pour copier les variables et les valeurs associées
  IloNumVarArray startVar(env);
  IloNumArray startVal(env);
  for (int i = 0; i < m; ++i)
  {
     for (int j = 0; j < n; ++j)
     {
         //2. copie
         startVar.add(x[i][j]);
         startVal.add(start[i][j]);
     }
}
   //3. appel de addMIPStart avec les tableaux 1D
   cplex.addMIPStart(startVar, startVal);

   //4. on signale a Cplex la fin d'utilisation des tableaux 1D
   startVal.end();
   startVar.end();
```

Exemple d'utilisation de la fonction `addMIPStart()` avec des tableaux 2D (extrait du manuel utilisateur [15])

Remarque 3 : La fonction `addMIPStart()` n'a bien sûr pas de sens pour les PL fractionnaires. Pour ces derniers il existe la fonction `setStart()`.

5 REFERENCES

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

[2] IBM, IBM Knowledge Center. Solving mixed integer programming problems (MIP). https://www.ibm.com/support/knowledgecenter/en/SSSA5P_12.6.3/ilog.odms.cplex.help/CPLEX/UsrMan/topics/discr_optim/mip/01_mip_title_synopsis.html

[3] IBM, IBM Knowledge Center, Tutoriel Concert Technology pour les utilisateurs du C++. http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.0/ilog.odms.cplex.help/CPLEX/GettingStarted/topics/tutorials/Cplusplus/cpp_synopsis.html?lang=fr

[4] 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

[5] IBM, IBM Knowledge Center, Coupes. http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.0/ilog.odms.cplex.help/CPLEX/UsrMan/topics/discr_optim/mip/cuts/26_cuts_title_synopsis.html?lang=fr

[6] IBM, IBM Knowledge Center, Fonctions d'optimisation des performances de l'optimiseur PMNE. http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.0/ilog.odms.cplex.help/CPLEX/UsrMan/topics/discr_optim/mip/performance/12_title_synopsis.html?lang=fr

[7] Emilie Danna, Edward Rothberg et Claude Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. Mathematical Programming, volume 102-1, pages 71–90. 2005.

[8] Edward Rothberg. An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS Journal on Computing, volume 19, issue 4, pages 534–541. 2007.

[9] Matteo Fischetti, Fred Glover, Andrea Lodi, The feasibility pump. Mathematical Programming, Springer. 2005.

[10] Livio Bertacco, Matteo Fischetti, Andrea Lodi, A feasibility pump heuristic for general mixed-integer problems, Discrete Optimization, Volume 4, Issue 1, Pages 63-76. 2007. ISSN 1572-5286.

[11] Tobias Achterberg, Timo Berthold, Improving the feasibility pump. Discrete Optimization. Volume 4, Issue 1, Pages 77-86. 2007. ISSN 1572-5286.

[12]IBM, IBM Support, FAQ. How do I select non default parameters to tune CPLEX's performance on a difficult mixed integer program? http://www-01.ibm.com/support/docview.wss?uid=swg21400023

[13] IBM, IBM Knowledge Center. Rapports d'avancement : interprétation du journal des nœuds. https://www.ibm.com/support/knowledgecenter/fr/SSSA5P_12.6.3/ilog.odms.cplex.help/CPLEX/UsrMan/topics/discr_optim/mip/para/52_node_log.html

[14]IBM, IBM ILOG CPLEX Optimization Studio CPLEX Parameters Reference, https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.3/ilog.odms.studio.help/pdf/paramcplex.pdf

[15]IBM, IBM ILOG CPLEX Optimization Studio CPLEX User's Manual, Version 12 Release 6, https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.3/ilog.odms.studio.help/pdf/usrcplex.pdf

[16]IBM, IBM Knowledge Center. Preprocessing: presolver and aggregator. https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.3/ilog.odms.cplex.help/CPLEX/UsrMan/topics/discr_optim/mip/para/48_preproc.html

[17]IBM, IBM Knowledge Center. Terminating MIP optimization
https://www.ibm.com/support/knowledgecenter/en/SSSA5P_12.6.3/ilog.odms.cplex.help/CPLEX/UsrMan/topics/discr_optim/mip/usage/11_terminate.html