A New Decoposition Method for Chromatic Number Problem

Khellouf leila <khelloufleila@yahoo.fr>

10 avril 2017

Table des matières

Résumé

Ce rapport présente le *branch and price*, technique utilisée en programmation linéaire pour résoudre des problèmes de grande taille et la librairie IBM qui implémente le *branch and price*. Ce dernier allie les techniques de *génération de colonnes branch and bound*.
Mots Clés : Branch and Price, Génération de Colonnes, IBM.

1 La Génération de Colonnes

La génération de colonnes est une technique utilisée pour résoudre un programme linéaire en nombres réels lorsque les variables sont trop nombreuses pour être toutes énumérées ou pour une résolution directe par un solveur. La génération de colonnes utilise, entre autres, la solution duale et les coûts réduits. On commence donc cette partie par un rappel concernant ces deux notions avant de présenter la génération de colonne proprement dite.

Principe

L'idée de la génération de colonnes - également appelée génération de variables : une colonne d'un programme linéaire correspondant à une variable - repose sur le fait que toutes les variables ne sont pas nécessaires pour trouver la solution optimale.

L'objectif est donc de résoudre le programme linéaire initial (appelé problème initial ou problème maitre) en prenant en compte un nombre restreint de variables mais suffisant.

Initialisation

Le PL est initialisé avec un nombre réduit de variables (choisies par une heuristique Dsatur). Ce PL (généré à partir d'un nombre restreint de variables) est alors appelé problème maitre restreint (PMR). A partir de là le but est de générer des variables améliorantes c'est-à-dire des variables qui n'ont pas été mise dans le PL initialement mais qui sont susceptibles d'améliorer l'objectif.

Génération des variables améliorantes : Le problème maitre restreint est résolu à l'optimal et on s'intéresse à la solution duale. Le coût des variables duales va nous permettre de trouver une (ou plusieurs) variables améliorantes.Le coût réduit d'une variable se calcule à l'aide de la solution duale de la manière suivante :

Coût réduit $\overline{c}_i$ d'une variable i.

$$\overline{c}_i = c_i - c_B.B^{-1}.A_i = c_i - u_B.A_i = c_i - \sum_{j=1}^{m} u_j a_{ji}$$

c_i : coefficient de la variable i dans la fonction objectif
u_j : variable duale associée à la contrainte j
a_{ji} : coefficient de la variable i dans la contrainte j
m : nombre de contraintes.
Les variables améliorantes sont celles qui ont un cout réduit strictement négatif(pour le problème de minimisation). L'idée est donc, une fois le problème maitre restreint résolu, de chercher les variables v qui n'ont pas encore été insérées dans le problème et qui ont un cout réduit négatif. Pour cela il est nécessaire de résoudre un autre PL(appelé *sous-problème* dans lequel les variables sont les a_{vj}(i.e. les coefficients de la nouvelle variables v dans la contrainte j) et les coefficients sont les couts des variables duales liées à la solution optimale du problème maitre restreint

2 L'algorithme de Branch and Price

Les algorithmes de branch and bound avec génération de colonnes sont appelés algorithmes de branch and price. Plus précisément dans un branch and price on a une exploration arborescente de la même façon que dans le branch and bound mais au lieu d'avoir une simple résolution de PL á chaque nœud on á résoudre un PL avec génération de colonnes. Les colonnes générées peuvent être valides dans tout l'arbre ou seulement dans la branche courante. Les deux principales difficultés dans la mise en œuvre d'un branch and price sont :

1. Formaliser le sous-problème et être capable de la résoudre rapidement (même difficulté que dans le cadre de la génération de colonnes) ;
2. Trouver une règle de branchement adaptée qui ne perturbe pas le sous problème (difficulté propre au branch and price).

3 Programmer un Branch and Price avec IBM

Dans cette partie on donne les principes généraux qu'on illustre avec des morceaux de code.

3.1 Le Pricer

On illustre cette partie par des extraits de code utilisé pour résoudre le PL par la génération de colonnes.

3.1.1 La Classe Column

Cette classe nous permet de sauvegarder toute les colonnes ajouter par la génération de colonnes .

```
public List<Column> Columns = new ArrayList<Column>();
```

```
public Column(List<Integer> addc)
element = new ArrayList<Integer>();
for (int i = 0; i < addc.size(); i++)
element.add(addc.get(i));
xi = Columns.size();
Columns.add(this);
public int a(int i)
return element.contains(i) ? 1 : 0;
```

3.1.2 La Classe Master Problem

Cette Classe nous permettre d'effectuer la génération dynamiques de variables. Il est associé à la classe **MasterProblem** qui a pour rôle de gérer la phase de génération de colonnes(ou pricing). Cette classe contient une méthode **createDefaultStables()** qui initialise le PL avec un nombre réduit de variables trouver par l'heuristique **DSATUR**

```
public MasterProblem() throws IloException
createModel();
createDefaultStables();
column-generation :P arameters :configureCplex(this);
```

```
private void createModel() throws IloException
cplex = new IloCplex();
reduced - cost = cplex.addMinimize();
for(int i = 0; i < G.V; i + +)
inegality.put(i, cplex :addRange(1 ;Double :MAX -V ALUE; "Col" + i));
```

```
public void addNewColumn(Column col)
try
IloColumn new-Column = cplex.column(reduced-cost, col.cost);
for(int i = 0; i < G.V (); i + +)
new Column = new Column.and(cplex.column(inegality.get(i); col.a(i)));
col.x = cplex. numVar(new Column; 0; 1; "S :" + col.xi);
catch (IloException e)
System :err :println("Concert exception caught : " + e);
```

3.1.3 La Classe SubProblem

Il y a généralement trois étapes dans la méthode :

1. Récupérer la valeur des variables duales pour pouvoir construire le sous problème.

2. Résoudre le sous-problème.
3. Utiliser la solution du sous-problème pour créer et ajouter au PL une nouvelle variable. Les contraintes doivent être mises á jour avec cette variable.

```
public SubProblem() throws IloException
this.constraints = new ArrayList<IloRange>() ;
createModel() ;
column-generation.Parameters.configureCplex(this) ;
```

```
private void createModel() throws IloException
cplex = new IloCplex() ;
x = new IloIntVar[G.V()] ;
for(int u = 0 ; u < G :V () ; u ++)
x[u] = cplex.intVar(0 ; 1) ;
reduced-cost = cplex.addMaximize() ;
for(int u = 0 ; u < G :V () ; u ++)
for(int w : G.adj(u))
constraints.add(cplex.addLe(cplex.sum(cplex.prod(1, x[u]), cplex.prod(1, x[w])),
1, "ct" + constraints.size())) ;
```

Le rôle de cette méthode est de chercher des variables améliorantes et de les ajouter au PL courant. Si aucune variable n'a été ajoutée alors **Cplex** considère qu'il n'y a plus de variable améliorante. Sinon **Cplex** met à jour la solution courante et les variables duales en prenant en compte la les nouvelle(s) variable(s). **updateReducedCost()** est appelée a chaque fois qu'on à mis à jour la solution fractionnaire courante(soit parce que on a branché, ou ajouter une nouvelle variables ou une nouvelle contrainte).

```
public void updateReducedCost() throws IloException, IOException
IloNumExpr num-exp = cplex.linearNumExpr() ;
num-exp = cplex.prod(x[0], 0) ;
for(int i = 0 ; i < G :V () ; i + +)
num-exp = cplex.sum(num-exp, cplex.prod(this.x[i], ColumGeneration.
this.masterproblem.pi.get(i))) ;
reduced-cost.clearExpr() ;
reduced-cost.setExpr(num-exp) ;
```

La méthode **Solve()** permet de résoudre la sous problème.

```
public void Solve()
try
if (cplex.solve())
if (1 - cplex.getObjValue() <= -0.00000001)
SaveColumn();
```

saveColumn nous permettre d'ajouter une nouvelle variable au RMP grâce à la solution du sous problème .

```
public void SaveColumn()
try
List<Integer> addc = new ArrayList<Integer>();
for (int i = 0; i < G.V(); i++)
if (cplex.getValue(x[i]) > 0.99999999)
addc.add(i);
ColumGeneration.this.masterproblem.addNewColumn(new Column(addc));
catch (IloException e)
System.out.println("....." + e);
```

3.2 Le Branchement

3.2.1 Implémenter un branchement de type Ryan et Foster

Principe

On a vu dans la partie théorique dédiée au Branch and price que les régles de branchement habituelles fonctionnent mal avec la génération de colonnes. on á donc présenté une régle (Ryan and Foster) plus adéquate.
Dans cette partie on a crée des noeud fils qui sont copie conforme de leur pére avec une contrainte supplémentaire (la contrainte de branchement).

Le choix des Sommets

Soit x^* une solution optimal primal fractionnaire dans un noeud de l'arbre de branchement (π^* sa solution dual correspondante). Sélectionnons deux sommets i et $j \in V$ tel que :

$$\sum_{s \in S, i,j \in s} x_s^* = \alpha,\ \alpha \text{ est fractionnaire.}$$

Dans les nœuds fils on doit :

1. Supprimer les sous ensembles de colonnes irréalisable(qui ne respect pas le branchement).
2. Ajouter les contraintes nécessaire pour le problème de pricing (Pour générer des colonnes avec les propriétés souhaitées).

3.2.2 Same

Cette partie consiste à regrouper deux sommets v_1, v_2 en un seul sommet w dans G. Un sommet quelconque de G a alors un arc vers w si et seulement s'il a un arc dans le graphe initial vers v_1 ou vers v_2.

```
 public void BranchementGauche(int v1, int v2) throws IloException
for (Column Col : Columns)
if (Col.a(v1)== 1 and Col.a(v2)== 0 || Col.a(v2)== 1 and Col.a(v1) == 0)
Col.x.setUB(0);
subproblem.constraints.add((IloRange)subproblem.cplex.
addEq(subproblem.cplex.prod(1, subproblem.x[v1]),
subproblem.cplex.prod(1, subproblem.x[v2]), "ctEquality" +
subproblem.constraints.size()));
```

3.2.3 Differ

Differ consiste à ajouter un arc entre v_1 et v_2, ils ne peuvent alors plus être dans le même stable par définition même d'un stable

```
public void BranchementDroit(int v1, int v2) throws IloException
for (Column Col : Columns)
if (Col.a(v1) == 1 and Col.a(v2) == 1)
Col.x.setUB(0);
subproblem.constraints.add(subproblem.cplex.addLe(
subproblem.cplex.sum(subproblem.cplex.prod(1, subproblem.x[v1]),
subproblem.cplex.prod(1, subproblem.x[v2])),1, "ctAddEdge" +
subproblem.constraints.size()));
```

3.2.4 Branch and Bound

Le déroulement du *branch and bound* est personnalisé par une classe *excution* et en rédéfinissant certaines méthodes.

```
public boolean execution() throws IloException, Exception
boolean testStop = false;
boolean debut = true;
while (!testStop)
if (debut)
boolean entier = BP.runColumnGeneration1();
if (entier)
borneSuperieur = (int) BP.masterproblem.LastObjValue;
return true;
BP.choixsommet();
int v1 = BP.sommet1;
int v2 = BP.sommet2;
Noeud nd = new Noeud(v1, v2, BP.masterproblem.LastObjValue, true);
Arbre.add(nd);
debut = false;
return testStop = true;
```

```
Else {
miseAJour();
boolean entier = BP.runColumnGeneration1();
boolean stop = comparaisonBornes();
if (entier || stop || BP.masterproblem.LastObjValue == -1)
{
if (entier and BP.masterproblem.LastObjValue < borneSuperieur
and BP.masterproblem.LastObjValue != -1)
borneSuperieur = (int) (BP.masterproblem.LastObjValue + 0.1);
while (!Arbre.isEmpty() and Arbre.get(Arbre.size() - 1).gauche == false
Arbre.remove(Arbre.get(Arbre.size() - 1));
if (Arbre.isEmpty())
return true;
Arbre.get(Arbre.size() - 1).gauche = false;
}
```

```
Else {
BP.choixsommet();
int v11 = BP.sommet1;
int v21 = BP.sommet2;
double lbd = BP.masterproblem.LastObjValue;
Noeud e = new Noeud(BP.sommet1, BP.sommet2, lbd, true);
Arbre.add(e);
if (Arbre.size() == 0)
testStop = true;
debut = false;}
```

```
 public void miseAJour() throws IloException
for (Column x : this.BP.Columns) {
x.x.setUB(1);
x.x.setLB(0);
}
for (int i = 0; i < this.Arbre.size(); i++)
if (this.Arbre.get(i).gauche)
BP.BranchementGauche(this.Arbre.get(i).u, this.Arbre.get(i).v);
else
BP.BranchementDroit(this.Arbre.get(i).u, this.Arbre.get(i).v);
```

```
public boolean comparaisonBornes()
if (borneSuperieur <= (int) (BP.masterproblem.LastObjValue + 0.9))
return true;
else
return false;
```