Introduction au *Branch Cut and Price* et au solveur SCIP (Solving Constraint Integer Programs)

Hélène Toussaint [1]

Rapport de recherche LIMOS/RR-13-07

19 avril 2013

1. helene.toussaint@isima.fr

Résumé

Ce rapport présente le *branch cut and price* (BCP), technique utilisée en programmation linéaire pour résoudre des problèmes de grande taille et la librairie **SCIP** qui implémente le *branch cut and price* sous forme de *framework*. Le *branch cut and price* allie les techniques de *branch and bound* , *branch and cut* et *branch and price*. Ces trois techniques sont donc présentées séparément et illustrées sur des exemples.

Mots clés : Programmation linéaire en nombres entiers, *Branch Cut and Price*, génération de colonnes, **SCIP**

Table des matières

1 Introduction

A qui s'adresse ce document ?

- aux personnes désirant se familiariser avec le *branch cut and price*
- aux personnes désirant découvrir le solveur **SCIP** (http ://scip.zib.de/) dans sa version *callable library*

Prérequis : Pour comprendre les parties théoriques de ce document le lecteur doit avoir des connaissances en programmation linéaire. Pour les parties techniques / programmation le lecteur doit avoir de bonnes connaissances en C++.

Qu'est ce que le *branch cut and price* (BCP) ? Le BCP allie les techniques d'exploration arborescente, génération de coupes et génération de colonnes pour résoudre un programme linéaire en nombre entier ou mixte. Nous reviendrons sur chacune de ces techniques plus en détail dans la suite du document.

Qu'est-ce que SCIP ? SCIP est un solveur pour PLNE (programme linéaire en nombre entier) ou problèmes mixtes. Il peut être utilisé en ligne de commande ou comme bibliothèque dans un programme en langage C ou C++. C'est cette deuxième utilisation à laquelle nous nous intéresserons dans ce document. SCIP est programmé comme un *framework* : il est découpé en modules ou *plugins* (module pour le branchement, pour la génération de colonnes, pour les coupes, pour les heuristiques primales, etc...) que l'utilisateur peut utiliser et paramétrer. De cette manière l'utilisateur peut intervenir sur chaque étape du processus de résolution (contrairement à Cplex par exemple qui s'utilise comme une "boîte noire"). De ce fait SCIP est tout à fait approprié lorsque l'on souhaite programmer un algorithme de résolution d'un PLNE basé sur la technique classique du *branch and bound* mais à laquelle on apporte des modifications (règles de branchement spécifiques, ajout de colonnes...).

NB : utilisé comme boîte noire sur un PLNE SCIP est plus lent que Cplex (environ 5 fois plus lent). Par contre il est très performant pour le codage d'un BCP spécifique.

But du document Ce document a pour but de présenter le *branch cut and price* d'un point de vue pratique (même si on rappelle de temps en temps des résultats théoriques nécessaires à la compréhension des concepts). On se focalise principalement sur les concepts, les cas d'utilisation des différentes méthodes (illustrées par des exemples) et sur les difficultés que l'on rencontre dans la mise en œuvre. On tente de donner une vue synthétique de la manière dont ces méthodes peuvent être implémentées à l'aide de la bibliothèque **SCIP** de sorte que le lecteur soit capable de prendre en main rapidement cette bibliothèque.

Structure du document On commence tout d'abord par présenter **SCIP** dans sa version "boîte noire" afin de se familiariser avec ce logiciel (section 2). La section 3 donne les clés pour comprendre le fonctionnement de **SCIP** dans sa version *callable*

library (en C++). La section 4 présente le *branch and bound* et la section 5 montre comment programmer cette méthode avec **SCIP**. La section 6 s'intéresse à la génération de colonnes car cette technique est utilisée par le *branch and price* présenté section 7 pour les concepts et section 8 pour la programmation. Enfin le *branch and cut* est présenté section 9 (concepts) et 10 (programmation). On termine par une synthèse sur le *branch cut and price*.

2 Utilisation basique de SCIP : les fonctions de base pour la résolution d'un PLNE

Dans cette partie on propose de découvrir les fonctions de base de SCIP. C'est à dire les fonctions qui permettent d'utiliser SCIP comme bibliothèque pour résoudre un PLNE sans intervenir sur le processus de résolution. Un certain nombre d'exemples et de documents sont disponibles sur le site de SCIP [1].

L'exemple le plus facile fournit avec SCIP est l'exemple des n-reines. Il contient une documentation en latex très claire dont certains passages sont repris ici. Les étapes pour résoudre un PLNE avec SCIP sont :

1. Initialiser l'environnement SCIP
2. Charger les plugins
3. Créer le problème
4. Ajouter les variables et contraintes
5. Résoudre le PL
6. Accéder aux résultats
7. Libérer l'environnement SCIP

Chacun de ces points est détaillé dans les sous-parties suivantes. Les exemples de cette partie ont été réalisés avec SCIP 2.0.1. Les fonctions de SCIP disposent souvent d'une longue liste d'arguments qui permet de régler les paramètres. On ne donne pas de descriptif de tous les arguments, ils sont disponibles sur le site de SCIP. Afin de mettre l'accent uniquement sur les fonctions de SCIP, nous allons illustrer chacune de ces étapes sur un PL très simple contenant deux variables et deux contraintes (Extrait de [2]) : la figure 1 montre la résolution graphique.

$$(P)\begin{cases} \text{Minimiser} & -8x_0 - 5x_1 \\ \text{sous} & \\ & x_0 + x_1 \leq 6 \\ & 9x_0 + 5x_1 \leq 45 \\ & x_0, x_1 \in \mathbb{N} \end{cases}$$

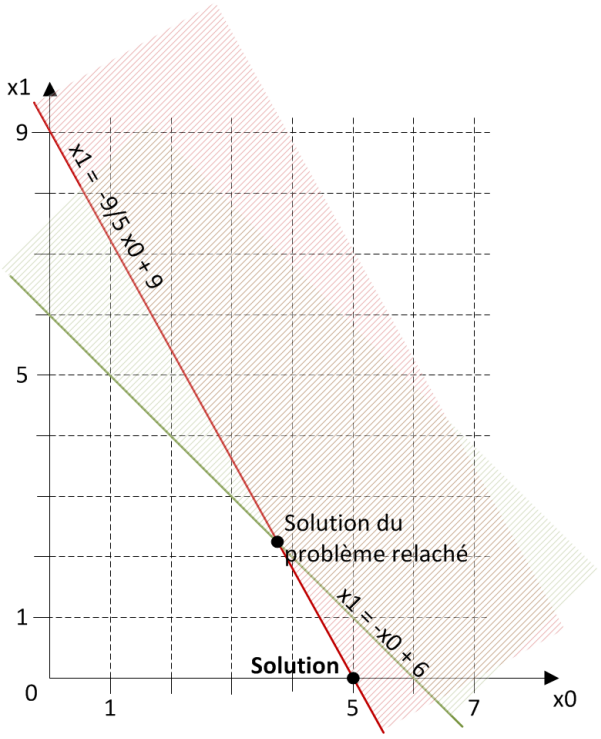

FIGURE 1 – Résolution graphique de (P)

2.1 Initialiser l'environnement SCIP

On initialise le pointeur sur l'environnement SCIP avec SCIPcreate() (figure 2).

```
//===========================================
// 1. creation de l'environnement SCIP
SCIP * scip;
SCIPcreate(&scip);
```

FIGURE 2 – Initialisation de l'environnement SCIP

2.2 Charger les plugins

Le fonctionnement de SCIP s'appuie sur l'utilisation de *plugins*. Après avoir initialisé l'environnement, on charge les *plugins* par défaut. Il existe de nombreux *plugins*, tous les ceux nécessaires sont inclus par la fonction SCIPincludeDefaultPlugins() (figure 3).

```
//============================================================
// 2. charger les plugins par défaut (gestions des ctr, heuristiques, ... )
SCIPincludeDefaultPlugins(scip) ;
// le chargement des plugins active l'envoie de messages sur la sortie standard
//pour désactiver :
SCIPsetMessagehdlr(NULL) ;
```

FIGURE 3 – Chargement des plugins

2.3 Créer le problème

Pour créer le PLNE proprement dit (c'est à dire la structure qui contiendra les variables et contraintes) on appelle SCIPcreateProb(). Dans l'exemple de la figure 4 on crée un PL vide, les variables et contraintes seront ajoutées par la suite.

```
//============================================================
// 3.creation du probleme
// 3.1 creer un probleme vide
SCIPcreateProb(scip, "LN", 0, 0, 0, 0, 0, 0, 0);
```

FIGURE 4 – Création d'un problème vide

Par défaut on minimise, si on souhaite changer de sens il suffit d'appeler la fonction SCIPsetObjSense().

```
//par défaut on crée un pb de minimisation si on souhaite maximiser :
SCIPsetObjsense(scip, SCIP_OBJSENSE_MAXIMIZE) ;
```

FIGURE 5 – Fixer le sens d'optimisation

2.4 Gérer les variables

La figure 6 montre comment :
- Créer les variables (SCIPcreateVar())
- Ajouter les variables dans le problème (SCIPaddVar())

Les arguments de SCIPcreateVar() sont :
- pointeur sur l'environnement **SCIP**
- adresse du pointeur sur la variable à ajouter au PL
- nom de la variable
- borne inf. de la variable
- borne sup. de la variable
- coefficient de la variable dans la fonction objectif
- type de la variable

– ... (les 7 derniers ne présentent pas d'intérêt ici et doivent être mis comme dans l'exemple)

```
// 3.2 ajouter des variables
SCIP_VAR * var1;
SCIPcreateVar(scip, &var1, "x0", 0, SCIP_REAL_MAX, -8,
  SCIP_VARTYPE_INTEGER, true, false, 0,0,0,0,0);
SCIPaddVar(scip,var1);

SCIP_VAR * var2;
SCIPcreateVar(scip, &var2, "x1", 0, SCIP_REAL_MAX, -5,
  SCIP_VARTYPE_INTEGER, true, false, 0,0,0,0,0);
SCIPaddVar(scip,var2);
```

FIGURE 6 – Création et ajout des deux variables au problème

NB : à partir de la version 3.0 de **SCIP** est disponible la fonction SCIPcreateVarBasic() qui place directement les 7 derniers arguments comme dans l'exemple.

2.5 Gérer les contraintes

```
// 3.3 ajouter des contraintes
SCIP_CONS * cons1;
SCIPcreateConsLinear(scip, &cons1,"c1", 0, 0, 0, -SCIPinfinity(scip), 6.0,
    TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE) ;
SCIPaddCoefLinear(scip, cons1, var1, 1);
SCIPaddCoefLinear(scip, cons1, var2, 1);
SCIPaddCons(scip, cons1) ;


SCIP_CONS * cons2;
SCIPcreateConsLinear(scip, &cons2,"c2", 0, 0, 0, -SCIPinfinity(scip), 45.0,
  TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE) ;
SCIPaddCoefLinear(scip, cons2, var1, 9);
SCIPaddCoefLinear(scip, cons2, var2, 5);
SCIPaddCons(scip, cons2) ;
```

FIGURE 7 – Création et ajout des contraintes au problème

Pour créer des contraintes il faut choisir une fonction de création de contrainte (il en existe une par type de contrainte). Pour les contraintes linéaires il s'agit de la fonction SCIPcreateConsLinear() qui gère les contraintes de la forme :

$$lhs \leq a^T x \leq rhs$$

– pour une contrainte égalité il faut mettre $lhs = rhs$

– pour une contrainte " inférieure " il faut mettre $lhs = -SCIPinfinity(scip)$
– pour une contrainte " supérieure " il faut mettre $rhs = SCIPinfinity(scip)$

La fonction SCIPaddCoefLinear() permet d'ajouter des variables existantes à une contrainte et SCIPaddCons() ajoute la contrainte au problème. La figure 7 montre la création et l'ajout des deux contraintes au problème (P).

2.6 Résoudre

La fonction SCIPsolve() lance la résolution du problème (figure 8).

```
//=============================================================
//4. resolution
SCIPsolve(scip);
```

FIGURE 8 – Résolution du problème

2.7 Récupérer la solution

On récupère un pointeur sur la solution optimale avec la fonction SCIPgetBestSol(). Ensuite les fonctions SCIPgetSolOrigObj() et SCIPgetSolVal() permettent d'accéder à la valeur de l'objectif et des variables (figure 9).

```
//=============================================================
//5. recuperation de la solution

SCIP_SOL * sol = SCIPgetBestSol(scip);
cout << "obj = " << SCIPgetSolOrigObj(scip,sol) << endl;
cout << "valeur x0= " << SCIPgetSolVal(scip, sol, var1) << endl;
cout << "valeur x1= " << SCIPgetSolVal(scip, sol, var2) << endl;
```

FIGURE 9 – Récupérer la solution optimale et afficher la valeur de la fonction objectif et la valeur des variables

2.8 Libérer la mémoire

```
//=============================================================
//6. libérer l'environnement

SCIPreleaseVar(scip, &var1);
SCIPreleaseVar(scip, &var2);
SCIPreleaseCons(scip, &cons1);
SCIPreleaseCons(scip, &cons2);
SCIPfree(&scip);
```

FIGURE 10 – Libérer les variables, les contraintes puis l'environnement

Une fois notre PLNE résolu et la solution récupérée on libère la mémoire :
– libération des variables et contraintes ;
– libération de l'environnement SCIP.

2.9 Remarques sur les macros SCIP_CALL

Si vous avez regardé les exemples fournis avec SCIP vous aurez remarqué l'utilisation de macros à l'appel des fonctions SCIP. Elles s'utilisent de la même façon : la fonction SCIP est passée entre parenthèses comme argument de la macro. Par exemple :

```
SCIP_CALL( uneFonctionScip() );
```

Il faut savoir que les fonctions **SCIP** retournent généralement un code erreur du type SCIP_RETCODE qui vaut 1 si tout c'est bien passé, 0 si l'erreur n'est pas identifiée et un entier négatif pour une erreur spécifique.

Les deux premières macros sont faites pour des programmes C : SCIP_CALL exécute la fonction qui lui est passée en paramètre et affiche un message d'erreur si la fonction a retourné une erreur ; SCIP_CALL_ABORT met fin au programme dans le cas où la fonction a retourné une erreur. SCIP_CALL_EXC est dédiée au langage C++ : elle génère une exception du type SCIPException dans le cas où la fonction a retourné une erreur :

```
try
{
  SCIP_CALL_EXC( uneFonctionScip() );
}catch(SCIPException & exc)
{
   cerr << exc.what() << endl;
   exit(exc.getRetcode());
}
```

L'utilisation de ces macros n'est pas obligatoire mais recommandée, cependant pour des raisons de lisibilité nous ne les mettons pas dans ce document.

2.10 Choix du solveur linéaire

SCIP utilise un solveur linéaire pour résoudre la relaxation du PLNE. Par défaut il utilise SoPlex (qui est fourni avec la suite **SCIP**) mais il peut être utilisé avec tous les solveurs qui possèdent une interface OSI (par exemple Cplex). Il suffit pour cela de *linker* avec le solveur de notre choix au moment de la compilation (voir la documentation d'installation sur le site de **SCIP**).

2.11 Remarque sur la résolution de problèmes en nombres réels avec SCIP

SCIP a été conçu pour résoudre des problèmes en nombres entiers ou mixtes. Il est tout à fait capable de résoudre des problèmes réels mais il n'est pas recommandé de l'utiliser pour cela parce que, d'une part les performances ne seront pas bonnes, et d'autre part on n'aura pas accès aux variables duales. Pour la résolution de problèmes réels on peut utiliser l'interface " lpi " fourni par SCIP. Il faut inclure le fichier d'entête scip/lpi.h. Nous ne nous étendrons pas sur cette interface, le but de ce document étant de résoudre des problèmes en nombres entiers.

3 Utilisation avancée de SCIP : présentation

3.1 Principe général

SCIP est programmé en module, chaque module jouant un rôle bien défini au cours du processus de résolution. Dans l'interface C++ de **SCIP** les modules sont des classes. Pour personnaliser le comportement d'un module il suffit d'utiliser l'héritage : on crée une classe fille à la classe que l'on souhaite personnaliser et on redéfinit les méthodes dont on souhaite modifier le comportement. Autrement dit, l'appel à SCIPsolve() lance le processus de résolution : ce processus peut être vu comme l'enchaînement d'un certain nombre de fonctions (ou méthodes) définies dans différentes classes. Lorsque l'utilisateur dérive une classe pour modifier le comportement d'une méthode le processus global de résolution reste inchangé à ceci près que la méthode redéfinie prend la place de la méthode de la classe mère dans le processus. Par exemple, imaginons qu'à un moment donné durant la résolution s'exécutent séquentiellement les méthodes f_1, f_2 d'une classe C_1 et g_1, g_2, g_3 d'une classe C_2 (figure 11 (a)). L'utilisateur souhaite modifier le comportement de la fonction f_2. Pour cela il crée une classe fille à la classe C_1, appelons là myC_1, et redéfinit la méthode f_2. Au lieu d'appeler $C_1 :: f_2$ SCIPsolve() fera alors appel à $myC_1 :: f_2$ (figure 11 (b)).

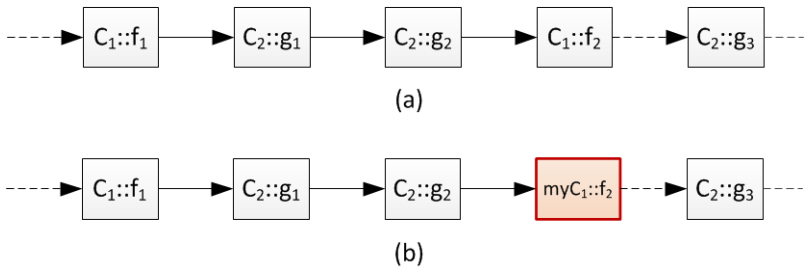

FIGURE 11 – (a) Séquence des appels par défaut – (b) Séquence des appels si l'utilisateur dérive myC_1 de C_1 et redéfinit uniquement la méthode f_2

Ce fonctionnement repose sur la notion d'héritage. Les classes de **SCIP** que l'on peut

dériver possède des méthodes virtuelles (c'est ce qui permet d'aller chercher dynamiquement la "bonne" méthode à exécuter). Certaines classes sont abstraites : elles possèdent des méthodes virtuelles pures (des méthodes virtuelles qui ne sont pas définies dans la classe initiale). C'est le cas des classes qui ne sont pas utiles dans le cadre de la résolution par défaut : par exemple la classe dont le rôle est d'ajouter des coupes (contraintes générées dynamiquement) en cours de résolution. En effet dans la résolution par défaut on suppose que toutes les contraintes sont initialement présentes dans le PLNE. Lorsque l'utilisateur souhaite dériver une classe abstraite il doit alors biensûr redéfinir toutes les méthodes virtuelles pures.

3.2 Les principales classes pour le *branch cut and price*

Il existe 18 classes dans **SCIP** que l'on peut dériver. On peut donc agir sur le processus de résolution très librement. Dans ce document on s'intéressera uniquement aux trois classes impliquées dans le cadre d'une résolution par *branch cut and price* .

NB : Il existe une interface de **SCIP** en langage C, les notions de classe et d'héritage n'existant pas en C l'utilisateur doit effectuer des modifications dans des structures contenues dans des fichiers spécifiques. Nous ne nous intéresserons pas, dans le cadre de ce document, à l'interface C.

4 Le *branch and bound*

Rappel : on se place dans le cadre d'une minimisation.

4.1 Principe général du *branch and bound*

Les algorithmes de *branch and bound* (B&B) sont des algorithmes de résolution exacte qui explorent l'espace des solutions en construisant un arbre de recherche. Ils utilisent la stratégie *diviser pour régner* en partitionnant l'espace de recherche des solutions en sous-problèmes pour les traiter de manière individuelle. Un sous-problème peut lui aussi être divisé en sous-problèmes de sorte que ce processus peut être vue comme la construction d'un arbre (ou comme une exploration arborescente).

Les algorithmes de B&B fonctionnent d'autant mieux si on connaît une borne supérieure et / ou une borne inférieure de bonne qualité au problème. Une borne supérieure peut être fournie par une heuristique : en effet une solution heuristique a un coût supérieur ou égal (*cas d'une minimisation*) au coût de la solution optimale. Une borne inférieure peut être fournie en relâchant certaines contraintes du problème, on optimise alors sur un ensemble de solutions réalisables plus large. La solution de ce problème a donc un coût inférieur ou égal au coût de la solution optimale.

Dans le cadre de ce document nous nous intéressons au B&B pour résoudre des programmes linéaires en nombres entiers (PLNE). Cependant cette stratégie peut être appliquée à des problèmes qui ne sont pas formalisés sous forme de PLNE.

4.2 Utilisation d'un B&B pour la résolution d'un PLNE

Cette partie décrit le déroulement d'un *branch and bound* pour la résolution d'un PLNE, elle s'appuie sur la documentation de COIN-OR/BCP ([5]). Le *branch and bound* est classiquement la technique utilisée pour résoudre un problème sous forme PLNE. L'idée est de relâcher les contraintes d'intégrité et de résoudre à l'optimal le problème en nombres réels (ce qui est beaucoup plus facile car on connaît des méthodes très performantes pour résoudre les PL réels comme la méthode du simplexe).

Le problème initial relâché constitue la racine de l'arbre. Chaque nœud de l'arbre est un PL réel et chaque branchement ajoute une (ou des) contraintes, qui portent généralement sur les variables, et qui tendent à rendre la solution entière. De cette manière le PL d'un nœud fils est constitué à partir du PL du père auquel on a ajouté une ou des contraintes. Les feuilles de l'arbre sont associées aux solutions entières du problème initial. La meilleure solution entière trouvée pendant l'exploration de l'arbre est la solution optimale.

Afin d'étudier ce processus plus en détail considérons $\mathcal{P}$ un PLNE. On note :
$\mathcal{S}$: l'ensemble des solutions de $\mathcal{P}$
s : une solution dans $\mathcal{S}$
$c(s)$: le coût de la solution s
s_b : la meilleure solution trouvée au cours du *branch and bound*

La solution s_b peut être initialisée par une heuristique. Au cours du *branch and bound* lorsqu'une meilleure solution est trouvée elle remplace s_b de sorte que cette variable contienne toujours la meilleure solution trouvée jusqu'à présent. Notons que $c(s_b)$ fournit une borne supérieure au coût de la solution optimale (*cas d'une minimisation*).

Durant le déroulement du *branch and bound* on garde à jour une liste $\mathcal{L}$ de sous-problèmes candidats ($\mathcal{L}$ contient donc les nœuds de l'arbre non encore explorés). Cette liste est initialisée avec le problème initial $\mathcal{P}$ dans lequel la contrainte d'intégrité est relâchée. A chaque étape on choisit un sous-problème $\mathcal{P}'$ dans $\mathcal{L}$, on le résout et on le supprime de $\mathcal{L}$. On note $\mathcal{S}'$ l'ensemble des solutions de $\mathcal{P}'$.

Il y a alors 4 possibilités :

1. Le problème P' est infaisable. Dans ce cas il est clair qu'on ne trouvera pas de solution dans cette branche. On élague.
2. La solution optimale du sous-problème P' n'est pas meilleure que $c(s_b)$. Dans ce cas aucune solution dans la branche courante ne pourra être meilleure que s_b. On élague.
3. La solution optimale du problème relâché est meilleure que $c(s_b)$ et elle est entière. Dans ce cas cette solution remplace s_b. De plus on ne pourra pas trouver mieux dans cette branche (on a déjà la solution optimale de la branche qui respecte toutes les contraintes), on élague.

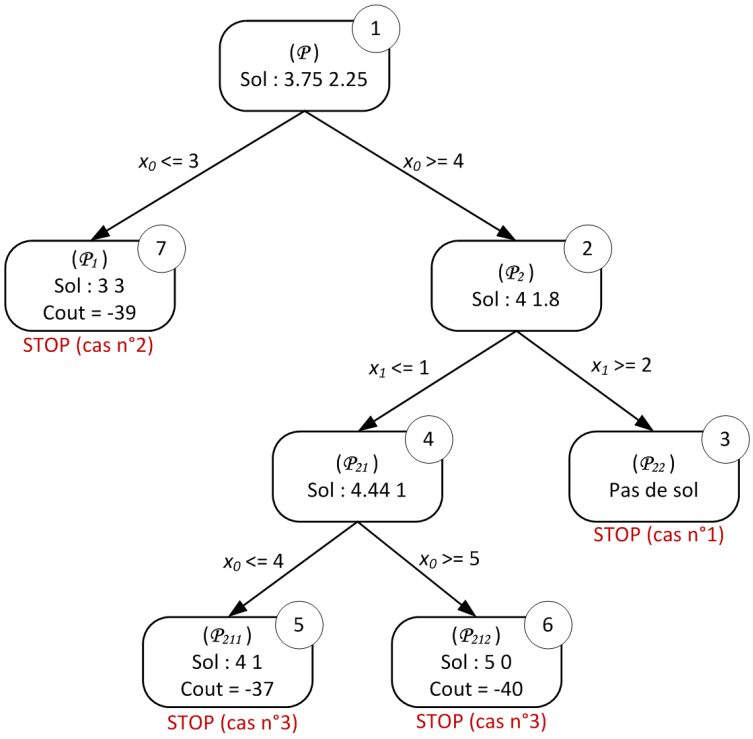

FIGURE 12 – Arbre de recherche associé à la résolution par *branch and bound* de $\mathcal{P}$

4. La solution optimale du problème relâché est meilleure que $c(s_b)$ mais elle n'est pas entière. Dans ce cas on branche : on identifie n sous-problèmes $\mathcal{P}'_1, \ldots, \mathcal{P}'_n$, (généralement $n = 2$) tel que l'union des ensembles des solutions de ces sous-problèmes forme $\mathcal{S}'$. Chacun de ces sous-problèmes est ajouté à $\mathcal{L}$.

Une fois un sous-problème traité (i.e. soit branché soit élagué) un nouveau sous-problème est choisi dans $\mathcal{L}$. Lorsque $\mathcal{L}$ devient vide s_b contient la solution optimale du problème initial.

4.3 Exemple

On considère le programme linéaire en nombre entier $(\mathcal{P})$ suivant (extrait de [2]) :

$$(\mathcal{P}) \begin{cases} \text{Minimiser} & -8x_0 - 5x_1 \\ \text{sous} & \\ & x_0 + x_1 \leq 6 \\ & 9x_0 + 5x_1 \leq 45 \\ & x_0, x_1 \in \mathbb{N} \end{cases}$$

Nous détaillons ici les différentes étapes de la résolution de ce problème par *branch and bound*. La Figure 12 montre l'arbre de recherche associé à cet exemple. Nous sup-

posons ne pas disposer d'une heuristique pour calculer une solution initiale. Nous considérons donc une borne supérieure initiale qui vaut $+\infty$.

Etape 1 $\mathcal{L} = \{\mathcal{P}\}$. On résout $\mathcal{P}$, on trouve $x_0 = 3.75$ et $x_1 = 2.25$. On décide (de manière arbitraire) de brancher sur x_0. On crée deux sous-problèmes $\mathcal{P}_1$ et $\mathcal{P}_2$ dans lesquels on fixe respectivement $x_0 \leq 3$ et $x_0 \geq 4$.

Etape 2 $\mathcal{L} = \{\mathcal{P}_1, \mathcal{P}_2\}$. On choisit de résoudre $\mathcal{P}_2$. On trouve $x_0 = 4$ et $x_1 = 1.8$. On branche sur x_1 car x_0 est déjà entier. Ce qui nous amène aux nouveaux sous-problèmes $\mathcal{P}_{21}$ et $\mathcal{P}_{22}$ qui ajoutent respectivement les contraintes $x_1 \leq 1$ et $x_1 \geq 2$ au problème $\mathcal{P}_2$.

Etape 3 $\mathcal{L} = \{\mathcal{P}_1, \mathcal{P}_{21}, \mathcal{P}_{22}\}$ On choisit de résoudre $\mathcal{P}_{22}$. Il n'y a pas de solution. On élague (cas n^o1).

Etape 4 $\mathcal{L} = \{\mathcal{P}_1, \mathcal{P}_{21}\}$ On choisit de résoudre $\mathcal{P}_{21}$. On trouve $x_0 = 4.44$ et $x_1 = 1$. On branche sur x_0 car x_1 est déjà entier. Ce qui nous amène aux nouveaux sous-problèmes $\mathcal{P}_{211}$ et $\mathcal{P}_{212}$ qui ajoutent respectivement les contraintes $x_0 \leq 4$ et $x_0 \geq 5$ au problème $\mathcal{P}_{21}$. Notons que comme on est déjà dans la branche $x_0 \geq 4$, la contrainte $x_0 \leq 4$ fixe en fait $x_0 = 4$.

Etape 5 $\mathcal{L} = \{\mathcal{P}_1, \mathcal{P}_{211}, \mathcal{P}_{212}\}$ On choisit de résoudre $\mathcal{P}_{211}$. On trouve $x_0 = 4$ et $x_1 = 1$. Les deux variables sont entières, cette solution est donc solution du problème initial (mais pas nécessairement optimale !). Le coût de cette solution est -37. On est dans le cas n^o3, on a trouvé une nouvelle meilleure solution (qui fournit donc une borne supérieure) et on élague.

Etape 6 $\mathcal{L} = \{\mathcal{P}_1, \mathcal{P}_{212}\}$ On choisit de résoudre $\mathcal{P}_{212}$. On trouve $x_0 = 5$ et $x_1 = 0$. Les deux variables sont entières, cette solution est donc solution du problème initial. Le coût de cette solution est -40. On est dans le cas n^o3.

Etape 7 $\mathcal{L} = \{\mathcal{P}_1\}$ On résout $\mathcal{P}_1$. On trouve $x_0 = 3$ et $x_1 = 3$. Le coût de cette solution est -39. On est dans le cas n^o2 : cette solution est plus mauvaise que la meilleure solution courante s_b donc ce n'est pas la peine de continuer, on élague.

Etape 8 La liste de sous-problèmes est vide : on a fini le parcours de l'arbre : la meilleure solution trouvée est donc la solution optimale de $\mathcal{P}$. Il s'agit de $x_0 = 5$ et $x_1 = 0$ d'un coût -40.

5 Programmer un *branch and bound* avec SCIP

Rappel : Le but de cette partie est de présenter l'utilisation de **SCIP** pour implémenter un *branch and bound*. On donne les principes généraux qu'on illustre avec

des morceaux de code mais on ne présente pas toutes les subtilités (voir les exemples disponibles sur le site de **SCIP** pour cela). De plus on se place dans le cadre d'une minimisation et d'un code C++.

5.1 Principe général

Comme on l'a vu dans la section précédente le branchement permet de *couper* la solution courante en divisant l'espace des solutions : le problème lié au nœud courant est divisé en sous-problèmes "plus petit". En général la solution courante n'est plus solution dans les sous-problèmes : on dit qu'elle est *coupée*. Si la solution n'est pas coupée il faut faire attention à ce que les sous-problèmes soient bien des restrictions du problème courant sinon le risque est de créer des sous-problèmes à l'infini.

Par défaut, l'appel à SCIPsolve() sur un PLNE réalise tout le processus de *branch and bound* sans que l'utilisateur n'ait besoin d'intervenir. Il utilise des règles par défaut (concernant par exemple le choix de la variable sur laquelle brancher, le nombre de sous-problèmes créés, le choix du prochain nœud à explorer...). Néanmoins **SCIP** offre à l'utilisateur la possibilité d'agir sur le déroulement de ce processus. Les prochaines sections montrent comment faire (en C++).

5.2 Déclaration d'une classe dédiée au branchement

Le déroulement du *branch and bound* peut être personnalisé en créant une classe fille à la classe de **SCIP** ObjBranchRule et en redéfinissant certaines méthodes (voir figure 13).

Le constructeur de la classe permet d'initialiser les paramètres suivant :

priority priorité de la règle de branchement, les règles de branchement sont appelées dans l'ordre décroissant de leur priorité jusqu'à ce qu'une des règles arrive à brancher ;

maxDepth profondeur maximale dans l'arbre de branchement à laquelle cette règle peut être appliquée, (-1 signifie illimité) ;

maxBoundDist distance relative maximale entre la borne duale du nœud courant et la borne primale comparée à la meilleure borne duale trouvée pour appliquer la règle de branchement (0 signifie que la règle est appliquée seulement au meilleur nœud courant et 1 signifie qu'elle est appliquée à tous les nœuds).

5.3 Exécution du branchement dans SCIP

L'exécution du branchement proprement dit : *i.e. la création des sous-problèmes (ou nœuds fils)* s'effectue dans la méthode scip_execlp() de la classe ObjBranchRule. C'est d'ailleurs la seule méthode virtuelle pure de ObjBranchRule.Cette méthode constitue le cœur du processus de branchement : au cours de la résolution elle est appelée au moment où il faut brancher sur la solution courante (classiquement au moment où le nœud

```
class MyBranchrule : public ObjBranchrule
{
public:

  MyBranchrule( SCIP* scip, const char* p_name, int priority, int maxDepth, double maxBoundDist );

  ~MyBranchrule( ){}

  virtual SCIP_RETCODE scip_execlp(
    SCIP*              scip,              /**< SCIP data structure */
    SCIP_BRANCHRULE*   branchrule,        /**< the branching rule itself */
    SCIP_Bool          allowaddcons,      /**< should adding constraints be allowed to avoid a branching? */
    SCIP_RESULT*       result             /**< pointer to store the result of the branching call */
    );
};
```

FIGURE 13 – Déclaration d'une classe pour le branchement

courant est résolu à l'optimal mais la solution n'est pas entière).

Dans la méthode scip_execlp() on crée les nœuds fils et on associe une (ou plusieurs) contraintes à ces fils en fonction de notre règle de branchement. Si on a une règle simple l'utilisation de cette méthode est suffisante pour gérer le branchement. Par contre si on a un branchement un peu plus compliqué (de type *Ryan and Foster* et / ou une phase de *pricing* par exemple) on aura besoin de stocker les décisions de branchement pour les propager correctement lorsque l'on parcourt l'arbre et / ou que l'on génère de nouvelles variables (on verra comment gérer cela dans la section concernant le *pricing* (8.3)).

5.4 Exemple de programmation d'une règle "simple" avec SCIP

On souhaite appliquer la règle suivante : on choisit la variable x la plus fractionnaire et on branche dessus créant ainsi deux nouveaux nœuds fils. Soit val la valeur courante de x. Le premier nœud fils sera associé à la contrainte $x \leq \lfloor val \rfloor$ et le second à la contrainte $x \geq \lceil val \rceil$. Notons que ce branchement est valide car il coupe bien la solution fractionnaire courante et on ne perd aucune solution entière (un des fils contient la solution optimale).

Les principales étapes pour réaliser ce branchement sont :

1. Déclarer les principales variables pour le branchement : les nœuds fils sont du type SCIP_NODE , les contraintes de branchement qui seront liées à chacun des nœuds fils sont du type SCIP_CONS (voir figure 14) ;
2. Récupérer les variables candidates au branchement et choisir celle sur laquelle on branche : la fonction SCIPgetLPBranchCands() fournit la liste des variables fractionnaires, l'utilisation de cette fonction est illustrée figure 15.
   NB : Si on souhaite utiliser un autre critère que la fractionnalité pour le branchement on peut stocker les variables dans une structure *ad hoc*, de cette manière on peut à tout moment connaître la valeur courante des variables en utilisant la fonction SCIPgetVarSol ;
3. Créer les nœuds fils à l'aide de la fonction SCIPcreateChild (figure 16) ;

4. Créer les contraintes de branchement et les associer aux nœuds fils (figure 17). Dans notre exemple les contraintes de branchement sont de simples contraintes linéaires, on peut donc utiliser la fonction SCIPcreateConsLinear() que l'on a déjà rencontrée pour la création d'un PL (voir section 2). Il faut cependant faire attention de bien mettre les *flags* local et stickingatnode à TRUE pour informer **SCIP** que les contraintes sont des contraintes locales liées aux nœuds et qu'il ne faut pas les ajouter de manière globale au PLNE.
5. Mettre à jour la variable **SCIP** result. Cette variable informe **SCIP** de ce qui a été fait dans la fonction : plusieurs valeurs sont possibles, dans notre cas on utilise SCIP_BRANCHED pour dire à **SCIP** que le branchement est réalisé (des nœuds fils ont été créés), voir figure 18.

```
SCIP_RETCODE MyBranchrule::scip_execlp(
  SCIP * scip,  SCIP_BRANCHRULE * branchrule, SCIP_Bool allowaddcons, SCIP_RESULT * result )
{

  //============================================================================================
  // 1. Déclaration des variables
  int nlpcands, npriolp;
  SCIP_VAR ** lpcands;
  double *    lpcandssol, * lpcandsfrac;
  SCIP_NODE * nodeMoins, * nodePlus;
  SCIP_CONS * consMoins, * consPlus;
```

FIGURE 14 – Déclaration de la fonction et des principales variables

```
//============================================================================================
// 2. on recupere les var. fractionnaires candidates au branchement
SCIPgetLPBranchCands(scip, &lpcands, &lpcandssol, &lpcandsfrac, &nlpcands, &npriolp);
cout << "nb de candidats au branchement = " << nlpcands << endl;

int ind = 0; //indice de la variables sur laquelle on veut brancher
double frac = 1;
for (int i = 0; i < nlpcands; ++i )
{
   cout << "fractionnalite de  "<< SCIPvarGetName(lpcands[i]) << " = " << lpcandsfrac[i] << endl;
   if ( abs(lpcandsfrac[i] - 0.5) < frac )//on choisit la plus fractionnaire (proche de 0.5)
   {
     ind = i;
     frac = abs(lpcandsfrac[i] - 0.5);
   }
}
// on crée un pointeur sur la variable candidate
SCIP_VAR* var = lpcands[ind];
cout << "branchement sur " << SCIPvarGetName(var) << "de cout = " << SCIPgetVarSol(scip,var) << endl;
```

FIGURE 15 – Récupération des variables fractionnaires

```
//=============================================================================================
// 3. on cree les noeuds enfants (ici deux, mais on pourrait en creer plus)
SCIPcreateChild(scip, &nodeMoins, 0.0, SCIPgetLocalTransEstimate(scip)) ;
SCIPcreateChild(scip, &nodePlus, 0.0, SCIPgetLocalTransEstimate(scip)) ;
```

FIGURE 16 – Construction des contraintes de branchement

```
//=============================================================================================
// 4. on associe les contraintes de branchement aux noeuds enfants
//4.1 créer les ctr
SCIPcreateConsLinear(scip, &consMoins, "cm", 0, 0, 0, -SCIPinfinity(scip), floor(SCIPgetVarSol(scip,var)),
  FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE, TRUE);
SCIPaddCoefLinear(scip, consMoins, var, 1);

SCIPcreateConsLinear(scip, &consPlus, "cp", 0, 0, 0, ceil(SCIPgetVarSol(scip,var)), SCIPinfinity(scip),
  FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE, TRUE);
SCIPaddCoefLinear(scip, consPlus, var, 1);

//4.2 ajouter les ctr aux noeuds
( SCIPaddConsNode(scip, nodeMoins, consMoins, NULL) );
( SCIPaddConsNode(scip, nodePlus, consPlus, NULL) );

//4.3 liberer les contraintes
( SCIPreleaseCons(scip, &consMoins) );
( SCIPreleaseCons(scip, &consPlus) );
```

FIGURE 17 – Association des contraintes aux nœuds

```
  //=============================================================================================
  //5. on met à jour la variable result : SCIP_BRANCHED si des noeuds enfants sont créés
  *result = SCIP_BRANCHED;

  return SCIP_OKAY;
}
```

FIGURE 18 – Mise à jour de la variable result

5.5 Comment programmer des règles de branchement complexes (par exemple Ryan & Foster) ?

Les règles de branchement plus complexes sont généralement utilisées dans le cadre du *branch and price*. Elles nécessitent de stocker des informations sur les nœuds et les variables impliquées dans le branchement. Une des manières de programmer ces règles avec **SCIP** est d'utiliser un *Constraint Handler*. Nous reviendrons plus en détail sur l'implémentation de telles règles dans la partie dédiée au *Branch and price* (section 7).

6 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 et de l'illustrer sur un exemple.

6.1 Rappels : problème dual et coûts réduits

Le but de cette section est de faire un court rappel sur des notions qui seront utiles pour comprendre la génération de colonnes. Elle s'appuie sur les premiers chapitres de [3].

6.1.1 Dualité

A tout problème linéaire on peut associer un second problème linéaire que l'on nomme problème dual (par opposition au premier qui est appelé problème primal).

Soit le problème primal ($\mathcal{P}$) suivant (n variables, m contraintes)

$$(\mathcal{P}) \begin{cases} \text{Minimiser} & \displaystyle\sum_{i=1}^{n} c_i x_i & \\ \text{sous} & & \\ & \forall j \in \{1, \ldots, m_1\}, & \displaystyle\sum_{i=1}^{n} a_{ji} x_i \geq d_j \\ & \forall j \in \{m_1+1, \ldots, m\}, & \displaystyle\sum_{i=1}^{n} a_{ji} x_i = d_j \\ & \forall i \in \{1, \ldots, n_1\}, & x_i \geq 0 \\ & \forall i \in \{n_1+1, \ldots, n\}, & x_i \text{ quelconque} \end{cases}$$

Le problème dual associé ($\mathcal{D}$) contient m variables, n contraintes.
Les variables u_j sont appelées variables duales. On remarque que :

- Le problème dual d'un problème de minimisation est un problème de maximisation (et vice versa). Dans la suite nous considérerons toujours un problème primal de minimisation (on peut toujours s'y ramener quitte à changer la fonction objectif de signe).

- A toute contrainte primale est associée une variable duale.

– A toute variable primale est associée une contrainte duale.

– Les coefficients de la fonction objectif du primal deviennent les seconds membres des contraintes du dual (et vice versa).

– une contrainte primale " $\geq$ " donne une variable duale positive ou nulle.

– une contrainte primale " $=$ " donne une variable duale libre.

– une variable primale positive ou nulle donne une contrainte duale " $\leq$ ".

– une variable primale libre donne une contrainte duale " $=$ ".

– la matrice des contraintes du dual est la transposée de la matrice des contraintes du primal.

$$(\mathcal{D}) \begin{cases} \text{Maximiser} & \displaystyle\sum_{j=1}^{m} d_j u_j & \\ \text{sous} & & \\ & \forall i \in \{1, \ldots, n_1\}, & \displaystyle\sum_{j=1}^{m} a_{ji} u_j \leq c_i \\ & \forall i \in \{n_1+1, \ldots, n\}, & \displaystyle\sum_{j=1}^{m} a_{ji} u_j = c_i \\ & \forall j \in \{1, \ldots, m_1\}, & u_j \geq 0 \\ & \forall j \in \{m_1+1, \ldots, m\}, & u_j \text{ quelconque} \end{cases}$$

Propriété de dualité faible Soit $\bar{x}$ une solution admissible du primal et $\bar{u}$ une solution admissible du dual. Alors on a :

$$c\bar{x} \geq \bar{u}d$$

Propriété de dualité forte Si le primal possède une solution optimale finie x^* alors le dual possède également une solution optimale finie u^* et on a :

$$cx^* = u^*d$$

6.1.2 Exemple primal / dual

Les deux tableaux de la figure 19 fournissent un exemple numérique. Le premier contient le problème primal (colonne 1), la solution optimale associée (colonne 2) et la solution duale associée (colonne 3). Le second contient le problème dual avec sa solution et sa solution duale.

Problème (primal)	Solution optimale	Solution duale
$\text{Min} \quad -x1 + 4x_2$ $\begin{cases} -x_1 \geq -7 \\ 3x_1 - 2x_2 \geq 4 \\ x_1 + x_2 \geq 5 \end{cases}$ $x_1 \geq 0,\ x_2$ libre	$x_1 = 7$ $x_2 = -2$ Objectif = -15	$u_1 = 5$ $u_2 = 0$ $u_3 = 4$

Problème dual	Solution optimale	Solution duale
$\text{Max} \quad -7u_1 + 4u_2 + 5u_3$ $\begin{cases} -u_1 + 3u_2 + u_3 \leq -1 \\ -2u_2 + u_3 = 4 \end{cases}$ $u_1 \geq 0,\ u_2 \geq 0,\ u_3 \geq 0$	$u_1 = 5$ $u_2 = 0$ $u_3 = 4$ Objectif = -15	$x_1 = 7$ $x_2 = -2$

FIGURE 19 – Exemple numérique primal / dual

On remarque que les deux problèmes ont effectivement la même valeur de la fonction objectif à l'optimum. De plus la solution duale du premier correspond à la solution du second et vice versa. On va voir dans la prochaine sous-section que la solution du dual se déduit directement de la solution du primal via les coûts réduits.

6.1.3 Coûts réduits

Sans entrer dans les détails (voir [3] pour les détails), rappelons nous que la résolution d'un PL par le simplexe entraine le choix d'une base B (matrice carrée inversible formée par un ensemble de colonnes de la matrice des contraintes A) de sorte que le problème :

Minimiser cx
sous
$Ax = d$
$x \geq 0$

se réécrit

Minimiser $c_B x_B + c_N x_N$
sous
$Bx_B + Nx_N = d$
$x_B \geq 0, x_N \geq 0$

où B est la base choisie, N la matrice qui " complète " B pour reformer la matrice initiale A, x_B les variables en base (qui correspondent aux colonnes choisies dans A pour former la base B) x_N les variables hors base, c_B les coûts dans la fonction objectif associée aux variables x_B et c_N ceux associés aux variables x_N.

Notons que les contraintes sont des contraintes égalités. Afin de transformer des contraintes inégalités en contraintes égalités on ajoute des variables d'écart : une variable d'écart par contrainte inégalité. Les variables d'écart sont toujours positives. On les ajoute donc négativement pour une contrainte " supérieure " et positivement pour une

contrainte " inférieure " (voir l'exemple figure 20).

Problème avec contraintes inégalité	Problème transformé : ajout de variable d'écart
Min $-x1 + 4x_2$ $\begin{cases} -x_1 \geq -7 \\ 3x_1 - 2x_2 \geq 4 \\ x_1 + x_2 \geq 5 \end{cases}$ $x_1 \geq 0$, x_2 libre	Min $-x1 + 4x_2$ $\begin{cases} -x_1 - x_3 = -7 \\ 3x_1 - 2x_2 - x_4 = 4 \\ x_1 + x_2 - x_5 = 5 \end{cases}$ $x_1 \geq 0$, x_2 libre, $x_3 \geq 0, x_4 \geq 0, x_5 \geq 0$

FIGURE 20 – Exemple numérique : Ajout de variables d'écart pour obtenir des contraintes égalités

Pour toute base B :

$$x_B = B^{-1}d \text{ et } x_N = 0$$

Lorsqu'on déroule le simplexe on utilise les coûts réduits pour savoir quelle variable doit entrer en base. Les coûts réduits $\bar{c_N}$ se calculent pour les variables hors base de la manière suivante :

$$\bar{c_N} = c_N - c_B.B^{-1}.N$$

Pour une variable i hors base quelconque (avec A_i la colonne i de A) :

$$\bar{c_i} = c_i - c_B.B^{-1}.A_i$$

Pour une variable d'écart j associée la $j^{ème}$ contrainte on a $c_j = 0$ (car une variable d'écart n'intervient pas dans l'objectif) et $A_j = e_j$ pour une contrainte $\leq$ ($-e_j$ pour une contrainte $\geq$), e_j étant le vecteur unité avec un 1 en $j^{ème}$ position et des 0 ailleurs. Il vient donc pour une variable d'écart j :

Pour la variable d'écart de la contrainte j (supposée $\leq$) :

$$\bar{c_j} = c_j - c_B.B^{-1}.A_j = 0 - c_B.B^{-1}.e_j = -c_B.B^{-1}$$

Signification : $\bar{c_j}$ donne "de combien" peut s'améliorer la fonction objectif si on relâche la contrainte j d'une unité.

A l'optimal on a : $c_B.B^{-1}.A - c \leq 0$ (voir [3] pour la démo).

Connaissant la base B associée à la solution optimale du primal peut-on en déduire la solution du dual ? Pour répondre à cette question revenons sur l'écriture du dual sous forme matricielle :

Problème primal
Minimiser cx
sous
$Ax = d$
$x \geq 0$

Problème dual
Maximiser ud
sous
$uA \leq c$
u libre

Posons $u_B = c_B.B^{-1}$ (avec u_B les variables duales en base).

1. u_B vérifie les contraintes :
   $u_B.A = c_B.B^{-1}.A$ et on sait que $c_B.B^{-1}.A - c \leq 0$ donc on a bien $u_B.A \leq c$
2. u_B maximise l'objectif :
   $u_B.d = c_B.B^{-1}.d = c_B.x_B$ (= valeur optimale !)

Finalement, si on connaît la base optimale B du primal la solution optimale du dual s'écrit :

$$u_B = c_B.B^{-1}$$

Or $c_B.B^{-1}$ est aussi l'expression du coût réduit des variables d'écart liées à des contraintes $\geq$:

- **Pour une contrainte $\geq$: la solution du dual est égale au coût réduit des variables d'écart**
- **Pour une contrainte $\leq$: la solution du dual est égale à l'opposé du coût réduit des variables d'écart**

Attention : lorsqu'on parle du *coût réduit d'une contrainte* on fait référence au coût réduit de la variable d'écart associée à la contrainte. Ce coût existe même si la variable d'écart n'a pas été explicitement introduite dans le PL ce qui peut prêter à confusion. En pratique on ne s'amuse jamais à mettre le PL sous forme standard (ie à introduire des variables d'écart) lorsqu'on résout un PL via un solveur. De ce fait pour récupérer le coût réduit d'une contrainte il faut demander au solveur la solution duale.

6.1.4 Exemple : calcul de coûts réduits

On reprend l'exemple précédent dans lequel on a ajouté des variables d'écart (figure 21). La figure 22 donne la base B et le coût des variables en base. La figure 23 montre les solutions optimales primales et duales calculées par Cplex pour le problème de la figure 21.

problème	solution	solution duale	coûts réduits
Min $-x1 + 4x_2$ $\begin{cases} -x_1 - x_3 = -7 \\ 3x_1 - 2x_2 - x_4 = 4 \\ x_1 + x_2 - x_5 = 5 \end{cases}$ $x_1 \geq 0$, x_2 libre, $x_3 \geq 0, x_4 \geq 0, x_5 \geq 0$	$x_1 = 7$ $x_2 = -2$ $x_4 = 21$	$u_1 = 5$ $u_2 = 0$ $u_3 = 4$	$\bar{c}_3 = 5$ $\bar{c}_4 = 0$ $\bar{c}_5 = 4$

FIGURE 21 – solution primale / duale

Les variables en base sont les variables non nulles : x_1, x_2 et x_4 sont en base.

$$B = \begin{pmatrix} -1 & 0 & 0 \\ 3 & -2 & -1 \\ 1 & 1 & 0 \end{pmatrix}; B^{-1} = \begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 1 \\ -5 & -1 & -2 \end{pmatrix}; \; c_B = (-1\,4\,0); \; c_B.B^{-1} = (5\,0\,4)$$

FIGURE 22 – Base B pour la solution optimale du problème figure 21 et coût des variables en base

```
Minimize
  obj: - x1 + 4 x2
Subject To
  c1: - x1 - x3 = -7
  c2: 3 x1 - 2 x2 - x4 = 4
  c3: x1 + x2 - x5 = 5
Bounds
  x2 Free
End
```

(a)

```
Dual simplex - Optimal:  Objective =   -1.5000000000e+01
Solution time =    0.00 sec.  Iterations = 0 (0)

CPLEX> dis sol dual -
Constraint Name            Dual Price
C1                           5.000000
C3                           4.000000
All other dual prices in the range 1-3 are zero.
CPLEX> dis sol reduced -
Variable Name            Reduced Cost
x3                           5.000000
x5                           4.000000
All other reduced costs in the range 1-5 are zero.
```

(b)

FIGURE 23 – (a) PL - (b) résultat affiché par Cplex

Pour plus de précisions sur la formulation d'un programme linéaire et la résolution par le simplexe on peut se référer à [3] qui propose à la fois des chapitres d'introduction détaillés et richement illustrés (formulation d'un problème, résolution graphique, algorithme du simplexe) et des chapitres plus spécialisés (entre autres : techniques de décomposition, méthode des points intérieurs, résolution en variables entières...).

6.2 La génération de colonnes

Attention ! La génération de colonne ne fonctionne que sur un problème en nombres réels car on a besoin des variables duales liées à la solution optimale du PL. Il n'y a pas de variables duales associées à la solution optimale d'un PLNE !

Quand utiliser la génération de colonnes ? Quand le PL possède potentiellement un très grand nombre de variables.

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 exemple de manière heuristique). 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. En effet on a vu dans la section sur la dualité la notion de coût réduit. Cette notion existe aussi pour des variables qui n'ont pas été introduites dans le PL. Le coût réduit d'une variable se calcule à l'aide de la solution duale de la manière suivante :

Coût réduit $\bar{c_i}$ d'une variable i :

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

Les notations sont celles de la section précédente, en particulier :
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 coût réduit strictement négatif (pour un 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 coût 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 variable v dans la contrainte j) et les coefficients sont les coûts des variables duales liées à la solution optimale du problème maitre restreint. Notons qu'il n'est pas nécessaire de faire apparaître le coefficient c_v (coefficient de la variable v dans la fonction objectif du problème initial) dans la fonction objectif du sous-problème car c'est une constante.

Forme générale du sous-problème :

Minimiser $c_v - \sum_{j=1}^{m} u_j a_{jv}$ $\left(\text{ou Maximiser } \sum_{j=1}^{m} u_j a_{jv}\right)$

sous " *la solution définie par les a_{jv} respecte les contraintes de variable du problème maître* "

Cette formulation du sous-problème nous fournit la variable qui possède le plus petit coût réduit. S'il est négatif alors la variable est améliorante et elle est ajoutée au problème maître restreint. Sinon cela signifie qu'il n'existe plus de variable améliorante, la solution du problème maitre restreint est la solution optimale du problème initial.

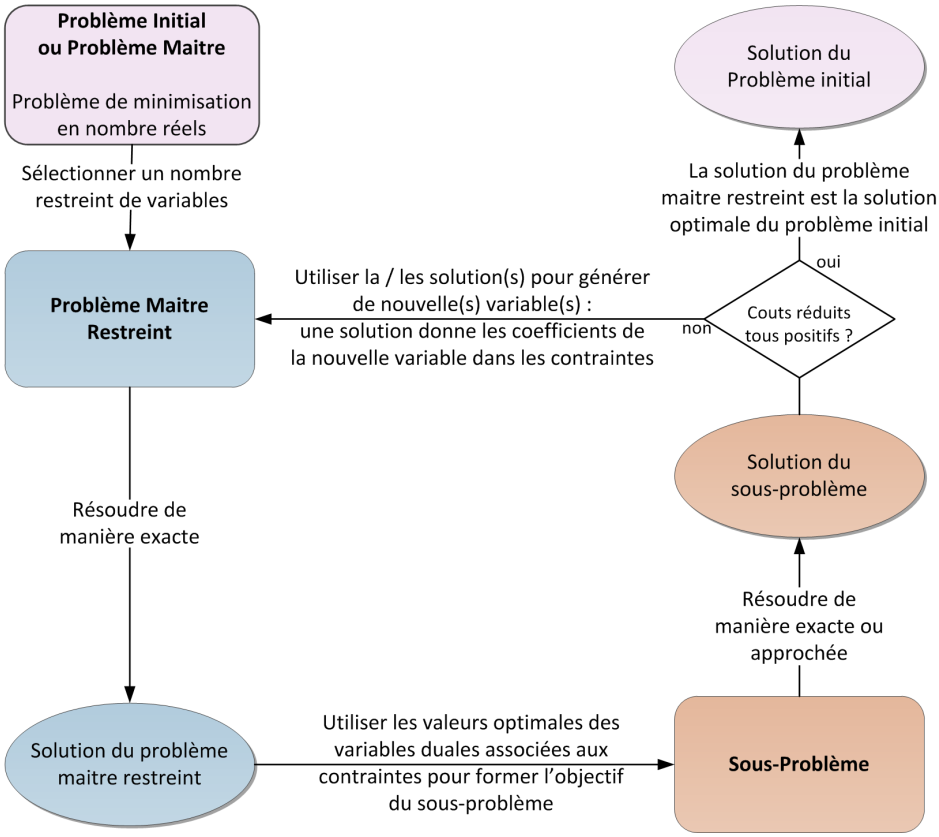

FIGURE 24 – Schéma général de la génération de colonnes

Si une variable améliorante est trouvée alors elle est ajoutée au problème maître restreint et le processus se poursuit : l'ajout de la nouvelle variable change la solution, on a donc de nouveaux coûts duaux donc un nouveau sous-problème à résoudre *etc.*

Notons que le sous-problème peut être résolu de manière heuristique. Cependant si l'heuristique ne donne pas de variable améliorante on est obligé de résoudre de manière exacte soit pour trouver une variable améliorante, soit pour prouver qu'il n'existe plus de variable améliorante.

La Figure 24 montre le mécanisme général de cette méthode. La sous-section suivante fournit un exemple numérique.

Mise en œuvre efficace de la génération de colonnes Il est bien évident que si le problème maitre restreint est de nouveau résolu entièrement à chaque ajout d'une nouvelle variable la génération dynamique de variables n'a plus aucun intérêt. Il ne faut pas repartir de zéro à chaque étape mais utiliser la solution optimale du PMR de l'étape précédente pour calculer la solution optimale du PMR à l'étape courante. En effet en

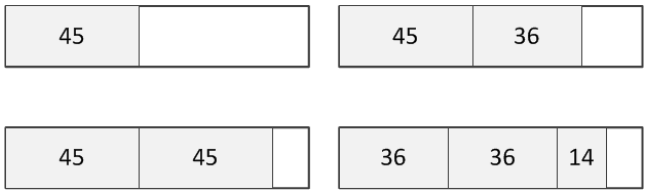

FIGURE 25 – Exemples de *patterns*

repartant de la dernière solution il y a de fortes chances que le simplexe n'ait que peu d'itérations à réaliser pour trouver la nouvelle solution.

6.3 Exemple : le problème de découpe en 1D (*Cutting stock problem*)

Cet exemple est extrait de [4].

Enoncé du problème Un atelier dispose de rouleaux de papier de 100m. Les clients demandent des bandes de papier de longueur $45m$, $36m$, $31m$ et $14m$ en quantités respectives 97, 610, 395 et 211. Il faut donc découper les rouleaux de $100m$ de sorte de satisfaire la demande tout en minimisant le nombre de rouleaux utilisés.

Construction du programme linéaire Il existe plusieurs manières possibles de découper un rouleau de 100 mètres pour obtenir des bandes de papier de longueur 45, 36, 31 et 14. Une bande de papier de longueur donnée est appelée *coupe*. Chaque manière de découper un rouleau est appelé *pattern*. Chaque *pattern* peut donc contenir un ou plusieurs coupes. La Figure 25 montre quelques exemples de *patterns* possibles. Le but est de trouver quels sont les *patterns* à découper et en quelle quantité pour satisfaire la demande tout en minimisant le nombre total de *patterns* (donc de rouleaux) utilisés.

Supposons que l'on ait calculé tous les *patterns* possibles. Le programme linéaire s'écrit alors :

$$\text{Minimiser} \sum_{i=0}^{n-1} x_i$$

sous

$$\forall j \in \{1, \ldots, 4\}, \sum_{i=0}^{n-1} a_{ij} x_i \geq d_j$$

$$x_i \in \mathbb{N}$$

avec
- x_i : nombre de *patterns* i utilisés
- n : nombre total de *patterns*
- a_{ij} : nombre de coupes j dans le *pattern* i
- d_j : demande en coupe j

Résolution du programme linéaire Nous ne souhaitons pas énumérer tous les *patterns* possibles. Nous allons donc nous orienter vers une approche par génération de colonnes : nous choisissons un nombre restreint de *patterns* qui forment une solution au problème (non optimale a priori). Dans le cas général nous pouvons utiliser une heuristique qui nous donnera une solution initiale de bonne qualité. Pour ce problème, qui a été choisi suffisamment simple pour trouver une solution initiale, nous choisissons trois *patterns* initiaux. Le premier (x_0) contient 1 coupe 36 et deux coupes 31. Le second (x_1) contient une coupe 45. Le troisième (x_2) contient une coupe 14.

La génération de colonnes s'appliquant à des problèmes en nombres réels, nous résolvons tout d'abord ce problème en nombre réels. Il est clair que d'un point de vue pratique la solution n'aura pas de sens (que signifie un nombre fractionnaire de *patterns* ?) mais cet exemple va permettre de comprendre comment fonctionne la génération de colonnes et elle sera le point de départ du *branch and price*, technique qui nous permettra de résoudre le problème en nombre entiers comme nous le verrons dans la section suivante.

Les quatre figures (26 à 29) donne le déroulement de la méthode. Le sous-problème peut être résolu de manière exacte (c'est ce qui est fait ici). Il peut être également résolu à l'aide d'une heuristique, ce qui est souvent le cas dans les problèmes de grande taille qui sont gourmands en temps de calcul.

(PMR_1)	Minimiser $x_0 + x_1 + x_2$ sous $\begin{cases} x_1 \geq 97 \\ x_0 \geq 610 \\ 2x_0 \geq 395 \\ x_2 \geq 211 \end{cases}$ x_0, x_1, x_2 *réels positifs*	Solution : $(x_0\ x_1\ x_2) = (610\ 97\ 211)$ Solution Duale : $(u_0\ u_1\ u_2\ u_3) = (1\ 1\ 0\ 1)$
(sousPb_1)	Maximiser $z_0 + z_1 + z_3$ sous $45z_0 + 36z_1 + 31z_2 + 14z_3 \leq 100$ z_0, z_1, z_2, z_3 *entiers positifs*	Solution : $(z_0\ z_1\ z_2\ z_3) = (0\ 0\ 0\ 7)$ cout réduit de la nouvelle variable : $\bar{c} = 1 - (0*1 + 0*1 + 0*0 + 7*1) = -6$ $\bar{c}$ négatif donc cette variable est améliorante

FIGURE 26 – Etape 1

(PMR_2)	Minimiser $x_0 + x_1 + x_2 + x_3$ sous $\begin{cases} x_1 \geq 97 \\ x_0 \geq 610 \\ 2x_0 \geq 395 \\ x_2 + 7x_3 \geq 211 \end{cases}$ x_0, x_1, x_2, x_3 *réels positifs*	Solution : $(x_0\ x_1\ x_2\ x_3) = (610\ 97\ 0\ 30.14)$ Solution Duale : $(u_0\ u_1\ u_2\ u_3) = (1\ 1\ 0\ 0.14)$
(sousPb_2)	Maximiser $z_0 + z_1 + 0.14z_3$ sous $45z_0 + 36z_2 + 31z_3 + 14z_4 \leq 100$ z_0, z_1, z_2, z_3 *entiers positifs*	Solution : $(z_0\ z_1\ z_2\ z_3) = (0\ 2\ 0\ 2)$ cout réduit de la nouvelle variable : $\bar{c} = 1 - (0 * 1 + 2 * 1 + 0 * 0 + 2 * 0.14)$ $= -1.28$ $\bar{c}$ négatif donc cette variable est améliorante

FIGURE 27 – Etape 2

(PMR_3)	Minimiser $x_0 + x_1 + x_2 + x_3 + x_4$ sous $\begin{cases} x_1 \geq 97 \\ x_0 + 2x_4 \geq 610 \\ 2x_0 \geq 395 \\ x_2 + 7x_3 + 2x_4 \geq 211 \end{cases}$ x_0, x_1, x_2, x_3, x_4 *réels positifs*	Solution : $(x_0\ x_1\ x_2\ x_3\ x_4) = (197.5\ 97\ 0\ 0\ 206.25)$ Solution Duale : $(u_0\ u_1\ u_2\ u_3) = (1\ 0.5\ 0.24\ 0)$
(sousPb_3)	Maximiser $z_0 + 0.5z_1 + 0.24z_2$ sous $45z_0 + 36z_2 + 31z_3 + 14z_4 \leq 100$ z_0, z_1, z_2, z_3 *entiers positifs*	Solution : $(z_0\ z_1\ z_2\ z_3) = (2\ 0\ 0\ 0)$ cout réduit de la nouvelle variable : $\bar{c} = 1 - (2 * 1 + 0 + 0 + 0) = -1$ $\bar{c}$ négatif donc cette variable est améliorante

FIGURE 28 – Etape 3

(PMR_4)	Minimiser $x_0 + x_1 + x_2 + x_3 + x_4 + x_5$ sous $\begin{cases} x_1 + 2x_5 \geq 97 \\ x_0 + 2x_4 \geq 610 \\ 2x_0 \geq 395 \\ x_2 + 7x_3 + 2x_4 \geq 211 \end{cases}$ $x_0, x_1, x_2, x_3, x_4, x_5$ *réels positifs*	Solution : $(x_0\ x_1\ x_2\ x_3\ x_4\ x_5) =$ $(197.5\ 0\ 0\ 0\ 206.25\ 48.5)$ Solution Duale : $(u_0\ u_1\ u_2\ u_3) = (0.5\ 0.5\ 0.25\ 0)$
(sousPb_4)	Maximiser $0.5z_0 + 0.5z_1 + 0.25z_2$ sous $45z_0 + 36z_2 + 31z_3 + 14z_4 \leq 100$ z_0, z_1, z_2, z_3 *entiers positifs*	Solution : $(z_0\ z_1\ z_2\ z_3) = (0\ 2\ 0\ 0)$ cout réduit de la nouvelle variable : $\bar{c} = 1 - (0 + 2 * 0.5 + 0 + 0) = 0$ $\bar{c}$ nul inutile de continuer, cette variable n'est pas améliorante

FIGURE 29 – Etape 4

La solution (réelle) du problème initial est 197.5 *patterns* x_0 ; 206.25 *patterns* x_4 et 48.5 *patterns* x_5.

7 Le *Branch and Price*

On reprend dans cette partie le vocabulaire utilisé pour la génération de colonnes (section 6).

7.1 Principe du *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 à chaque nœud on a à 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 :

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

7.2 Les difficultés liées au branchement dans un *branch and price*

En général les branchements classiques (typiquement le branchement binaire sur une variable) fonctionnent assez mal dans le cadre d'une résolution par *branch and price* . En effet dans un *branch and price* on est susceptible de générer de nouvelles variables à chaque nœud et par conséquent de faire augmenter rapidement le nombre de variables. Le risque est donc de parvenir difficilement à obtenir des solutions entières (qui correspondent aux feuilles de l'arbre) puisqu'on a régulièrement de nouvelles variables sur lesquelles brancher. De plus les contraintes de branchement viennent grossir l'ensemble des contraintes présentent initialement dans le PLNE. Elles sont biensûr associées à de nouvelles variables duales qui doivent être prises en compte dans le sous-problème (au moment où on cherche les variables améliorantes). Il n'est souvent pas facile de prendre en compte ces contraintes. La section 7.4 propose une solution classique : la règle de branchement *Ryan and Foster* et l'illustre sur un exemple.

7.3 Implémentation efficace d'un *branch and price*

Outre la difficulté de trouver une "bonne" règle de branchement et de gérer l'arborescence, il y a deux points délicats à traiter lorsque l'on programme un *branch and price* :

- la résolution du sous-problème ;
- la résolution du PL relâché (ou problème maître restreint) après l'ajout d'une contrainte de branchement (on entre dans un nouveau nœud : on a besoin de mettre à jour la solution courante) ou après l'ajout d'une nouvelle variable (on est dans le nœud courant : on a besoin de mettre à jour les variables duales pour

savoir s'il reste des variables primales améliorantes).

Résolution du sous-problème En ce qui concerne la résolution du sous-problème une stratégie est d'avoir une ou plusieurs heuristiques capables de générer rapidement des variables améliorantes et une méthode exacte qui s'assure qu'il n'existe plus de variables améliorante (ou qui en génère une) lorsque les heuristiques échouent. Cependant si on ne cherche qu'une solution approchée il est suffisant d'utiliser uniquement des heuristiques pour la recherche de la variables améliorante. Par contre il n'y a, dans ce cas, aucune garantie sur la qualité de la solution : on cumule à chaque nœud de l'arbre des imprécisions qui impactent les bornes supérieures et inférieures (solution duale). Au terme de l'exploration arborescente on peut donc avoir une différence assez importante entre le coût de la solution optimale et celui de la solution obtenue avec une génération de colonnes heuristique.

Résolution du problème maître restreint Le problème maître restreint est résolu au nœud racine (*via* un solveur linéaire qui utilise le simplexe) puis à chaque ajout de variable ou contrainte. L'ajout d'une contrainte ou d'une variable ne change pas en général complètement la solution. En effet, d'un point de vue géométrique la nouvelle solution a de forte chances d'être "proche" de l'ancienne dans le polyèdre convexe décrit par l'ensemble des solutions du PL courant (même si ce polyèdre est légèrement modifié par l'ajout d'une nouvelle contrainte). Cela signifie pour l'algorithme du simplexe qu'il n'a que peu d'itérations à exécuter pour accéder à la nouvelle solution en partant de l'ancienne (une itération du simplexe permet de passer d'un sommet du polyèdre à un sommet voisin, autrement dit d'une solution à une solution voisine). Il est donc primordial, au moins du point de vue des performances, de ne pas relancer la résolution du PL à chaque nœud ni à chaque ajout de variable mais de poursuivre la résolution en cours. Pour cela il faut avoir sauvegardé en mémoire l'état courant du simplexe. Il est clair que c'est fastidieux à programmer "à la main", heureusement il existe des librairies qui le font pour nous. C'est bien sûr le cas de la librairie **SCIP** présentée dans ce document (la section 8 sera dédiée à l'utilisation de **SCIP** pour le *branch and price*).

7.4 Exemple : *branch and price* pour le *graph coloring*

On présente dans cette section la règle de branchement *Ryan and Foster* bien connue dans le cadre de résolution d'un PLNE par *branch and price*. On illustre sur un exemple de coloration de graphe (*graph coloring*) extrait de [6].

7.4.1 Présentation du problème

Etant donné un graphe non orienté $G = (V, E)$, le problème de coloration de graphe consiste à attribuer une couleur à chaque sommet $v \in V$ du graphe de telle sorte que deux sommets u et v dans X reliés par une arête $e = (u, v) \in E$ soient de couleur différente. Une manière de formaliser ce problème par programme linéaire est la suivante : on note

S l'ensemble des stables maximaux de G, chaque stable est associé à une couleur. On cherche alors à couvrir tous les sommets de G par des stables de S tout en minimisant le nombre de stables choisis (donc le nombre de couleurs). On note x_s la variable binaire qui vaut 1 si le stable s est choisi, on obtient le PL $(\mathcal{P})$ suivant :

$$(\mathcal{P}) \begin{cases} \text{Minimiser} & \sum_{s \in S} x_s \\ \text{sous} & \\ & \forall v \in V, \quad \sum_{s \in S | v \in S} x_s \geq 1 \\ & \forall s \in S, \quad x_s \in \{0,1\} \end{cases}$$

Remarque : avec cette formulation un sommet peut appartenir à plusieurs stables (donc avoir plusieurs couleurs). Il suffit de choisir une couleur parmi celles qui lui sont attribuées. On peut aussi choisir de travailler avec des stables non maximaux mais il y en a alors beaucoup plus !

7.4.2 Résolution par *branch and price*

Pour un graphe quelconque il existe un très grand nombre de stables maximaux. L'idée est donc d'utiliser initialement un petit nombre de stables et de générer des stables améliorants par génération de colonnes. Pour écrire le sous-problème de la génération de colonne on considère la relaxation de $(\mathcal{P})$: x_s réel positif. Notons λ_v la valeur de la variable duale associée à la contrainte concernant le sommet $v \in V$. Plus λ_v est grand plus on a intérêt à faire entrer le sommet v dans le nouveau stable. On cherche donc un ensemble de sommet $v \in V$ qui constituent un stable ($\forall (v,u) \in E$ u et v ne peuvent pas être ensemble dans le stable) et qui maximise la somme des λ_v. Ce sous-problème est formalisé par $(S\mathcal{P})$ dans lequel z_v est la variable binaire qui vaut 1 si v est choisi, 0 sinon :

$$\text{Sous probleme } : (S\mathcal{P}) \begin{cases} \text{Maximiser} & \sum_{v \in V} \lambda_v z_v \\ \text{sous} & \\ & \forall (u,v) \in E, \quad z_u + z_v \leq 1 \\ & \forall v \in V, \quad z_v \in \{0,1\} \end{cases}$$

Si la solution de $(S\mathcal{P})$ vaut plus de 1 alors le stable (défini par l'ensemble des z_v égaux à 1) est potentiellement améliorant. Sinon il n'existe pas de stable améliorant. Dans ce dernier cas, si tous les x_s de $(\mathcal{P})$ sont entiers on a la solution optimale de $(\mathcal{P})$, sinon il faut brancher.

Pourquoi il ne faut pas utiliser un branchement binaire classique sur les variables ? Imaginons qu'une variable x_s soit fractionnaire et qu'on souhaite brancher dessus. On crée donc 2 problèmes fils : un dans lequel la variable est fixée à 0 et un autre dans lequel elle est fixée à 1 (x_s est binaire). Dans le premier problème fils on interdit l'utilisation du stable s. Comment faire passer cette information au sous-problème $(S\mathcal{P})$? Que se passe t-il si le meilleur stable (au sens du sous-problème $(S\mathcal{P})$) a été fixé à 0 dans la branche courante ? Comment alors savoir s'il existe un autre stable améliorant ? La solution serait de chercher le meilleur stable, puis le deuxième meilleur si le premier est interdit dans la branche, puis le troisième si le second est interdit dans la branche etc... Ceci serait très coûteux. En ce qui concerne le second fils (celui où on fixe x_s à 1) on impose le choix de s dans toute la branche. Cela n'est pas difficile à prendre en compte mais est très contraignant. On a donc deux problèmes fils très déséquilibrés : le premier a un ensemble de solutions proche de celui du problème père ($x_s = 0$ n'est pas très contraignant car il existe un très grand nombre de stables) alors que le second a un ensemble de solutions beaucoup plus petit.

Cette règle de branchement n'est donc pas du tout adaptée dans le cadre d'un *branch and price*.

La solution proposée dans [6] : la règle de *Ryan and Foster* Considérons une solution fractionnaire de $(\mathcal{P})$. Il est clair qu'il doit exister deux stables s_1 et s_2, et deux sommets u et v tels que $u \in s_1 \cup s_2$ et $v \in s_1 - s_2$ et au moins une des deux variables x_{s_1} ou x_{s_2} est fractionnaire. Ceci signifie que u et v sont ensemble dans le stable s_1 mais pas ensemble dans le stable s_2 ce qui est idiot : soit ils sont ensemble (ils ont la même couleur) soit non mais les deux à la fois n'est pas possible. On crée alors la règle de branchement suivante : "soit u et v sont dans le même stable soit ils sont dans des stables différents". Il est clair que cette règle conservent toutes les solutions entières et coupe le point fractionnaire considéré. Pour mettre en œuvre cette règle on crée les problèmes fils suivants :

Same(u, v)
Differ (u, v)

Same(u, v) consiste à regrouper u et v 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 u ou vers v. Differ (u, v) consiste à ajouter un arc entre u et v, ils ne peuvent alors plus être dans le même stable par définition même d'un stable. Le branchement s'applique donc sur l'instance et n'ajoute pas de contrainte linéaire au PL. De cette manière il n'y a pas de contraintes supplémentaires dans les problèmes fils donc pas problème pour les prendre en compte dans $(S\mathcal{P})$, pas de problème non plus avec de nouvelles variables duales éventuelles. De plus on garde exactement le même sous-problème $(S\mathcal{P})$ il est simplement appliqué à un graphe différent du graphe initial. Cette règle de branchement (très connue) est appelée règle de *Ryan and Foster*.

8 Programmer un *branch and price* avec SCIP

Rappel : Le but de cette partie est de présenter l'utilisation de **SCIP** pour implémenter un branch and price. On donne les principes généraux qu'on illustre avec des morceaux de code mais on ne présente pas toutes les subtilités (voir les exemples disponibles sur le site de **SCIP** pour cela). De plus on se place dans le cadre d'une minimisation et d'un code C++.

8.1 Le *pricer* : définition

Dans **SCIP** l'objet permettant d'effectuer la génération dynamique de variables s'appelle *pricer*. Il est associé à la classe abstraite ObjPricer qui a pour rôle de gérer la phase de génération de colonnes (ou *pricing*). Cette classe contient une méthode virtuelle pure : scip_redcost(). Lorsque l'on souhaite mettre en œuvre la génération de colonnes il faut donc dériver une classe fille de ObjPricer et redéfinir au moins la méthode scip_redcost(). Son rôle est de chercher des variables améliorantes et de les ajouter au PL courant. Si aucune variable n'a été ajoutée alors **SCIP** considère qu'il n'y a plus de variable améliorante. Sinon **SCIP** met à jour la solution courante et les variables duales en prenant en compte la/les nouvelle(s) variable(s). scip_redcost() est appelée à chaque fois qu'on a mis à jour la solution fractionnaire courante (soit parce qu'on a branché, ou ajouté une nouvelle variable ou une nouvelle contrainte).

Remarque : Même si SCIP a été conçu pour résoudre des PLNE, le *pricer* peut être utilisé dans le cadre d'une simple génération de colonnes sur un PL.

8.2 Le *pricer* : utilisation

On illustre cette partie par des extraits de code utilisés pour résoudre le PL de la section 6 par génération de colonnes.

8.2.1 Initialiser les contraintes dans le cadre de la génération de colonnes

On a vu dans la section 2 comment initialiser un PL. Lorsqu'on utilise un *pricer* les contraintes initiales sont susceptibles d'être modifiées en cours de résolution : en effet les nouvelles variables sont intégrées aux contraintes existantes. Pour que **SCIP** accepte d'intégrer de nouvelles variables dans des contraintes existantes, ces contraintes doivent avoir été créées avec le *flag* modifiable = TRUE dans la fonction SCIPcreateCons().

8.2.2 Dériver la classe ObjPricer

Comme on l'a vu dans la section 3 on modifie le comportement de **SCIP** au moment de la résolution en utilisant l'héritage. Si on souhaite inclure une phase de génération de colonnes il faut donc créer un classe fille à ObjPricer (voir figure 30). On peut ajouter les attributs que l'on souhaite à notre classe fille. On ajoute souvent un tableau de pointeurs sur les contraintes qui nous permettra de récupérer facilement les variables

duales liées aux contraintes au moment de la recherche de variables améliorantes. Dans notre exemple on a aussi ajouté un tableau de pointeurs sur les variables et un tableau contenant les coefficients du PL.

```
class ObjPricerDecoupe : public ObjPricer
{
  //attributs perso (facultatifs)
  int _nbVar ;
  vector< SCIP_VAR* >   _var;
  vector< SCIP_CONS* >  _cons;
  vector <vector<int> > _coeff;
```

FIGURE 30 – Attribut de la classe dérivée de ObjPricer (facultatif)

La méthode scip_redcost() est virtuelle pure dans ObjPricer, elle doit donc être obligatoirement redéfinie. Il faut aussi redéfinir scip_init() si on autorise **SCIP** à effectuer la phase de prétraitement (cas général). En effet par défaut **SCIP** effectue des transformations sur le PL (ou PLNE) qui ont pour but de le simplifier et donc d'accélérer la résolution. Cette méthode permet de récupérer les pointeurs sur les variables et contraintes transformées. Enfin la méthode scip_farkas() peut être appeler par le *pricer* dans le cas où le PL courant n'est pas faisable (c'est le cas par exemple si on n'a pas initialisé le PL avec suffisamment de variables ou si une contrainte de branchement a rendu le PL fils infaisable). La figure 31 montre les méthodes virtuelles que l'on a redéfinie dans notre classe fille.

```
//constructor
 ObjPricerDecoupe( SCIP* scip, const char* p_name,
   vector<SCIP_VAR*>& var, vector<SCIP_CONS*>& cons, vector <vector<int> > & coeff);

 //destructor
 virtual ~ObjPricerDecoupe(){}

 //give transformed variables and constraints
 virtual SCIP_RETCODE scip_init   ( SCIP* scip, SCIP_PRICER* pricer );

 //compute a new improving varaible - called if current LP is feasible
 virtual SCIP_RETCODE scip_redcost( SCIP* scip, SCIP_PRICER* pricer,
                                    SCIP_Real* lowerbound, SCIP_RESULT* result );

 //compute a new variable - called if current LP is not feasible
 virtual SCIP_RETCODE scip_farkas ( SCIP* scip, SCIP_PRICER* pricer );
```

FIGURE 31 – Méthodes de la classe dérivée de ObjPricer

8.2.3 Informer SCIP que l'on utilise un *pricer*

Par défaut **SCIP** n'intègre pas la phase de génération de variables lors de la résolution d'un PLNE. Il faut donc l'informer que l'on souhaite ajouter cette étape à notre résolution. Pour cela il faut inclure le *pricer* : SCIPincludeObjPricer() et l'activer : SCIPactivatePricer() après avoir fait les initialisations habituelles (initialiser l'environnement, le

problème...) mais bien sûr avant l'appel à SCIPsolve() (figure 32).

```
//=============================================================
//   Add a pricer

//creation du pricer
ObjPricerDecoupe* pricer_ptr = new ObjPricerDecoupe( scip, "pricerDecoupe",
                                                     var, cons, coeff );
//inclusion du pricer dans le projet
SCIPincludeObjPricer(scip, pricer_ptr, true);

//activation du pricer
SCIPactivatePricer(scip, SCIPfindPricer(scip, "pricerDecoupe"));
```

FIGURE 32 – Déclaration et inclusion d'un pricer (avant d'appeler SCIPsolve())

8.2.4 Redéfinir scip_init()

La méthode scip_init() est appelée une seule fois par **SCIP** avant la résolution proprement dite du PL (ou PLNE). Son rôle est de récupérer les pointeurs sur les variables et contraintes transformées de sorte que cette transformation soit transparente pour l'utilisateur dans la suite du processus (c'est-à-dire que l'utilisateur pourra continuer d'utiliser les pointeurs sur les contraintes et variables initiales comme si elles n'avaient pas été transformées). Pour cela on utilise SCIPgetTransformedVar() et SCIPgetTransformedCons() (voir figure 33).

```
SCIP_RETCODE ObjPricerDecoupe::scip_init ( SCIP* scip, SCIP_PRICER* pricer )
{
  for ( int i = 0; i < 3 ; ++i )
    SCIPgetTransformedVar ( scip, _var[i], &_var[i] );

  for ( int i = 0; i < 4 ; ++i )
    SCIPgetTransformedCons( scip, _cons[i], &_cons[i] );

  return SCIP_OKAY;
}
```

FIGURE 33 – Redéfinir scip_init()

8.2.5 Redéfinir scip_redcost() et scip_farkas()

Il y a généralement trois étapes dans la méthode scip_redcost() :

1. Récupérer la valeur des variables duales pour pouvoir construire le sous-problème (figure 34) ;
2. Résoudre le sous-problème ;
3. Utiliser la solution du sous-problème (s'il y en a une) pour créer et ajouter au PL une nouvelle variable. Les contraintes doivent être mises à jour avec cette variable (figure 35).

```
//=========================================
// get the dual solution
double lambda[4];

for ( int i = 0; i < 4; ++i )
  lambda[i] = SCIPgetDualsolLinear( scip, _cons[i] );
```

FIGURE 34 – Récupération de la valeur des variables duales

```
//creation de la nouvelle variable
SCIP_VAR* var;
_nbVar++;
stringstream name;    name << "x" << _nbVar;

SCIPcreateVar(scip, &var, name.str().c_str(), 0.0, SCIPinfinity(scip), 1,
                                SCIP_VARTYPE_INTEGER, false,  false, 0, 0, 0, 0, 0 ) ;

// mise a jour des contraintes
for (int i = 0; i < 4; ++i)
  SCIPaddCoefLinear(scip, _cons[i], var, solSousPb[i]) ;

// ajout de la variable dans SCIP
SCIPaddPricedVar(scip,  var,  1.0) ;

// sauvegarde du pointeur dans notre structure ad hoc
_var.push_back(var);
```

FIGURE 35 – Ajout d'une nouvelle variable grâce à la solution du sous-problème (solSousPb)

En ce qui concerne scip_farkas() la démarche est exactement la même que pour scip_redcost() sauf qu'on n'utilise pas les valeurs des variables duales. En effet scip_farkas() est appelé si le PL courant n'a pas de solution, il n'existe donc pas de solution duale non plus. On utilise à la place les valeurs de Farkas que l'on peut récupérer à l'aide de la fonction SCIPgetDualfarkasLinear().

8.2.6 Synthèse

Le schéma de la figure 36 montre le déroulement général de la phase de *pricing* : on considère le PL lié au nœud courant (si on est en nombres entiers il s'agit de la relaxation du PLNE courant), il est résolu à l'optimal par le solveur linéaire choisi (par défaut Soplex). S'il y a une solution scip_redcost() cherche des variables améliorantes sinon scip_farkas() cherche des variables qui tendent à rendre le PL faisable (lemme de Farkas). Les variables duales sont mises à jour par **SCIP** à chaque changement dans le PL.

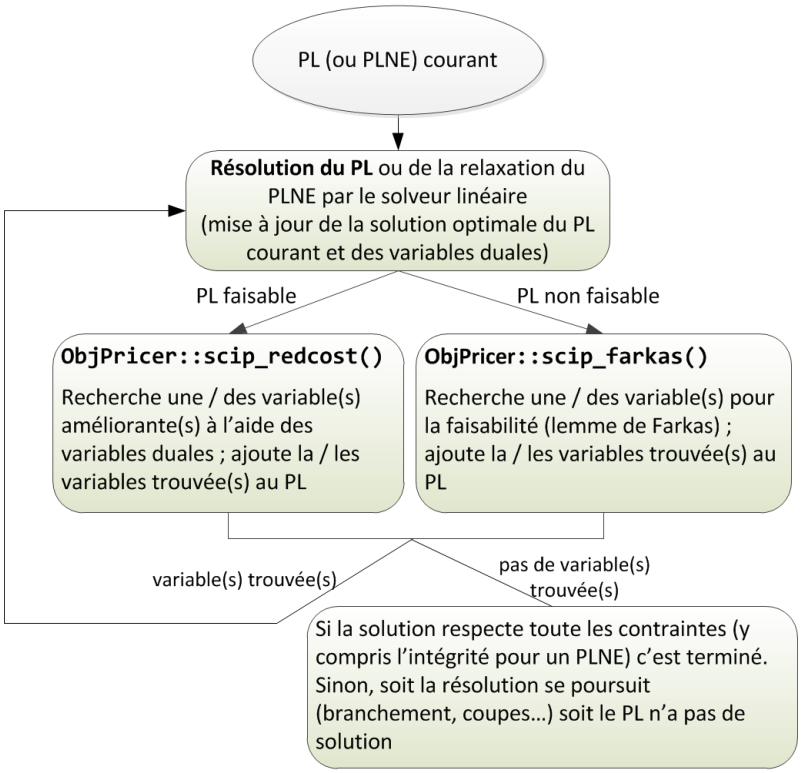

FIGURE 36 – La phase de *pricing*

8.3 Implémenter un branchement de type *Ryan and Foster*

8.3.1 Principe

On a vu dans la partie théorique dédiée au *branch and price* que les règles de branchement habituelles fonctionnement mal avec la génération de colonnes. On a donc présenté une règle (*Ryan and Foster*) plus adéquate. Comment peut-on programmer ce type de règle dans **SCIP** ?

Pourquoi ne peut-on pas utiliser ObjBranchRule de la même façon que pour une règle "simple" (ce qui a été présenté section 5) ? Dans la section 5 on crée des nœud fils qui sont la copie conforme de leur père avec une contrainte supplémentaire (la contrainte de branchement). Le problème lorsqu'on a une phase de *pricing* c'est que justement on souhaite éviter d'ajouter des contraintes supplémentaires car elles créent des variables duales qui perturbent le sous-problème. Il faut donc trouver un moyen de stocker les décisions de branchement sans les mettre explicitement sous forme de contrainte linéaire dans les PL fils.

La solution proposée par SCIP Pour résoudre ce problème **SCIP** dispose d'un gestionnaire de contraintes (*Constraint Handler*). L'idée est de stocker pour chaque

nœud les contraintes spécifiques qui lui sont associées (et éventuellement des données auxiliaires) dans le *Constraint Handler* au lieu de les ajouter dans le PL sous forme de contraintes linéaires. A chaque fois que l'on entre ou sort d'un nœud au cours du processus de résolution, certaines méthodes du *Constraint Handler* sont exécutées. Il suffit donc à l'utilisateur de redéfinir ces méthodes et de s'en servir pour faire les modifications appropriées au nœud courant.

8.3.2 Gestion de l'arborescence

Afin de bien comprendre comment utiliser le *constraint handler* on a besoin de savoir comment **SCIP** gère son arborescence. Tout d'abord **SCIP** numérote chaque nœud. La racine est le nœud numéro 1, puis les nœuds sont numérotés dans l'ordre où ils sont créés : le nœud n est le $n$ième nœud créé (peu importe l'ordre dans lequel l'arbre est parcouru). Par défaut **SCIP** ne réalise pas un parcours de l'arbre en profondeur mais explore en priorité le nœud le plus "prometteur", il réalise donc des "sauts" dans l'arbre et peut passer d'une branche à une autre très éloignée dans l'arbre. Pour cela il garde en mémoire les informations dont il a besoin pour chaque nœud non encore exploré (ce qui peut représenter un très grand nombre de données). Le type de parcours arborescent réalisé par **SCIP** dépend des *plugins* que l'on a chargés, l'appel à SCIPincludeDefaultPlugins() charge les *plugins* par défaut et entraîne donc le comportement par défaut de **SCIP**. Si on souhaite modifier ce comportement il suffit de charger uniquement les *plugins* que l'on souhaite. Par exemple le parcours de l'arbre dépend des *plugins* de type "*NodeSel*" (sélection de nœud), si on souhaite un parcours en profondeur il faut inclure uniquement le plugin correspondant : SCIPincludeNodeselDfs(). On peut aussi charger les *plugins* par défaut et changer les priorités à l'aide de la fonction SCIPsetIntParam() avant l'appel à SCIPsolve().

8.3.3 Implémentation

La classe qui implémente le *Constraint Handler* s'appelle ObjConshdlr. Elle contient différentes méthodes qui permettent de gérer différents types de contraintes (c'est aussi cette classe que l'on utilise pour le *branch and cut*). Dans le cas qui nous intéresse ici (stocker les décisions de branchement) les méthodes que l'on utilise sont :

- scip_active() : **activation** d'un nœud (appelée à chaque fois qu'on entre dans un nœud) ;
- scip_deactive() : **désactivation** (appelée à chaque fois qu'on quitte un nœud sauf si on le quitte pour aller dans un de ses fils) ;
- scip_prop() : **propagation** des contraintes liées au nœud courant (appelée quand on entre dans un nœud pour la première fois ou quand le nœud a été marqué pour être repropagé).

Pour bien comprendre à quel moment sont appelées ces fonctions supposons que SCIP construise l'arbre de la figure 37 et le parcourt en largeur : le processus qui mène à cet arbre est le suivant :

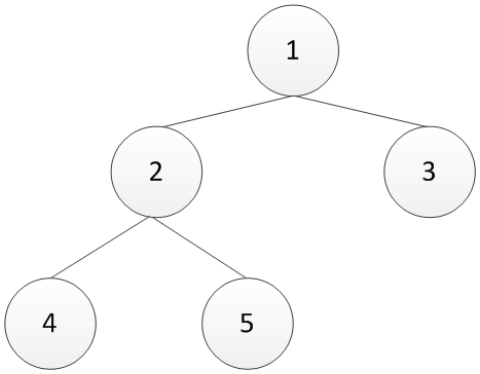

FIGURE 37 – Exemple d'arbre

- création du nœud 1 par l'utilisateur (problème initial) ; **activation** du nœud 1 ; **propagation** pour le nœud 1 ; résolution du nœud 1 ; branchement : création de deux fils au nœud 1 numéroté 2 et 3 ;
- sélection du nœud 2 ; **activation** du nœud 2 ; **propagation** pour le nœud 2 ; résolution du nœud 2 ; branchement : création de deux fils au nœud 2 numéroté 4 et 5 ;
- sélection du nœud 3 ; **désactivation** du nœud 2 ; **activation** du nœud 3 ; **propagation** pour le nœud 3 ; résolution du nœud 3 ; pas de fils (soit la solution est entière soit elle est coupée) ;
- sélection du nœud 4 ; **désactivation** du nœud 3 ; **activation** du nœud 2 ; **activation** du nœud 4 ; **propagation** pour le nœud 4 ; résolution du nœud 4 ; pas de fils ;
- sélection du nœud 5 ; **désactivation** du nœud 4 ; **activation** du nœud 5 ; **propagation** pour le nœud 5 ; résolution du nœud 5 ; pas de fils ; le parcours est terminé.

Remarque : La deuxième fois que le nœud 2 est activé il n'est pas repropagé car il a déjà été propagé une fois et **SCIP** a gardé en mémoire l'état du nœud après propagation (variables fixées, bornes ...). Toutefois si on souhaite que le nœud soit de nouveau propagé (par exemple on souhaite faire de nouvelles mise à jour sur les données) il suffit d'appeler la fonction SCIPrepropagateNode() dans scip_active() et le nœud sera repropagé après chaque activation (voir figure 38 : la classe fille de ObjConshdlr est appelée ConsBranchHdlr).

L'utilisateur peut avoir besoin de stocker des informations liées aux contraintes qu'il gérent grâce au *constraint handler*. Pour cela **SCIP** propose de créer une structure qu'il stocke avec la contrainte associée dans le *constraint handler* (la figure 39 donne un exemple de structure). Chaque méthode du *constraint handler* fournit dans ses arguments un pointeur sur la contrainte associée au nœud courant. L'utilisateur peut alors récupérer les données de la contrainte dont il a besoin par le biais de la fonction SCIP-

```
//scip_active est appele quand on rentre dans un noeud
SCIP_RETCODE ConsBranchHdlr::scip_active ( SCIP * scip,
  SCIP_CONSHDLR * conshdlr,
  SCIP_CONS * cons
  )
{
  SCIP_CONSDATA* consdata;
  consdata = SCIPconsGetData(cons);

  cout << " active noeud " << consdata->node->number << endl;

  //repropagé le noeud a chaque activation
  SCIPrepropagateNode(scip, consdata->node);

  return SCIP_OKAY;
}
```

FIGURE 38 – Exemple de redéfinition de scip_active()

```
//on va devoir ajouter des donnees aux contraintes
//on cree une  structure pour les stocker...
struct SCIP_ConsData
{
  int                   type;
  SCIP_NODE*            node;
  int                   couple1;
  int                   couple2;
};
```

FIGURE 39 – Création d'une structure pour stocker les informations d'une contrainte gérée par le *constraint handler*

consGetData comme cela a été fait dans l'exemple de la figure 38.

Les autres méthodes de ObjConshdlr (à savoir scip_check, scip_enfolp, scip_enfops, scip_sepalp, scip_sepaps) ne sont pas utiles pour gérer les décisions de branchement mais doivent quand même être redéfinies car elles sont virtuelles pures. Il suffit de les laisser vide, on retourne simplement SCIP_FEASIBLE comme si les contraintes étaient toujours satisfaites.

8.3.4 Exemple d'implémentation d'une règle de branchement *Ryan and Foster*

Cette section illustre la mise en œuvre d'un branchement *Ryan and Foster* sur l'exemple de la coloration de graphe de la partie 7.4 (il est nécessaire de l'avoir lue pour la compréhension de cette section). On ne présente ici que les fonctions principales. Le code complet est fourni sur le site de **SCIP** ([1]), il est implémenté en langage C mais cela ne change pas le fonctionnement global décrit ici.

Le fichier branch_coloring.c implémente le branchement proprement dit (création des fils, association d'une contrainte de branchement à chaque fils) et le fichier consStoreGraph.c implémente le *constraint handler* (stockage, activation des décisions de branchement).
Dans le fichier branch_coloring.c la méthode SCIP_DECL_BRANCHEXECLP (version C de scip_execlp()) cherche le couple de sommets qui va servir pour le branchement, crée les nœuds fils, les contraintes de branchement à l'aide d'une fonction *ad hoc* (COLORcreateConsStoreGraph : crée la contrainte et remplit la structure associée) et associe les contraintes aux nœuds fils (voir figure 40).

```
//recherhe de node1, node2 : les sommets du graphe qui serve pour le branchement
//   [ ... ]

// creation des fils (noeud vide pour le moment)
SCIP_CALL( SCIPcreateChild(scip, &childsame, 0.0, SCIPgetLocalTransEstimate(scip)) );
SCIP_CALL( SCIPcreateChild(scip, &childdiffer, 0.0, SCIPgetLocalTransEstimate(scip)) );

// creation des contraintes same et differ
currentcons = COLORconsGetActiveStoreGraphCons(scip);
SCIP_CALL( COLORcreateConsStoreGraph(scip, &conssame,   "same",
                      currentcons, COLOR_CONSTYPE_SAME,   node1, node2, childsame) );
SCIP_CALL( COLORcreateConsStoreGraph(scip, &consdiffer, "differ",
                      currentcons, COLOR_CONSTYPE_DIFFER, node1, node2, childdiffer) );

// ajout des contraintes aux noeuds
SCIP_CALL( SCIPaddConsNode(scip, childsame, conssame, NULL) );
SCIP_CALL( SCIPaddConsNode(scip, childdiffer, consdiffer, NULL) );
```

FIGURE 40 – Extrait de SCIP_DECL_BRANCHEXECLP pour le *graphe coloring* - le code complet est disponible sur le site de **SCIP** ([1])

Dans le fichier consStoreGraph.c la fonction SCIP_DECL_CONSACTIVE (version C de scip_active()) crée le graphe associé au nœud courant si cela n'a pas déjà été fait et marque le nœud courant pour être repropagé si de nouvelles variables ont été créées par la génération de colonnes depuis la dernière visite du nœud. La fonction SCIP_DECL_CONSPROP (version C de scip_prop()) propage les décisions de branchement qui ont été stockées par le *constraint handler* : on récupère tout d'abord les données liées à la contrainte (sommets du graphe impliqués dans la contrainte et type de la contrainte) et on fait les mises à jour nécessaires sur le PL :

- si la contrainte est du type DIFFER : les sommets liés à la contrainte ne doivent pas être dans le même stable. On fixe alors à 0 toutes les variables correspondantes à des stables qui contiennent les deux sommets (figure 41) ;
- si la contrainte est du type SAME : les sommets liés à la contrainte doivent être dans le même stable. On fixe alors à 0 toutes les variables correspondantes à des stables qui contiennent un seul des deux sommets.

```
//on recupere les donnees de la contrainte de branchement du noeud courant
cons = conshdlrData->stack[conshdlrData->nstack-1];
consdata = SCIPconsGetData(cons);

//cas 1 : contrainte de type DIFFER : toutes les variables associee a un stable contenant
// a la fois les sommets 1 et 2 sont fixees a 0
if (consdata->type == COLOR_CONSTYPE_DIFFER)
{
   for ( i = 0; i < nsets; i++ )
   {
      if ( !SCIPisFeasZero(scip, SCIPvarGetUbLocal(COLORprobGetVarForStableSet(scip, i))) )
      {
         if ( COLORprobIsNodeInStableSet(scip, i, consdata->node1)
           && COLORprobIsNodeInStableSet(scip, i, consdata->node2) )
         {
            var = COLORprobGetVarForStableSet(scip, i);
            SCIP_CALL( SCIPchgVarUb(scip, var, 0) );
            propcount++;
         }
      }
   }
}
```

FIGURE 41 – Extrait de SCIP_DECL_CONSPROP pour le *graphe coloring* - le code complet est disponible sur le site de **SCIP** ([1])

9 Le *branch and cut*

Le *branch and cut* (en français *coupes et branchements*) est un *branch and bound* avec génération dynamique de contraintes (souvent appelées *coupes* car elles servent à *couper* le point fractionnaire courant). On peut le voir comme la version duale du *branch and price* mais il est beaucoup plus simple à mettre en œuvre car les branchements binaires classiques sur les variables ne posent pas de problème dans le cadre d'un *branch and cut* ni les nouvelles variables duales liées à l'ajout des nouvelles contraintes. Il existe deux cas principaux d'utilisation du *branch and cut* :

1. on a une formulation PLNE et on souhaite utiliser des coupes pour éviter au maximum de brancher (cela étant très coûteux) en améliorant la valeur de relaxation,
2. on a une formulation PLNE avec un très grand nombre de contraintes et on ne souhaite pas (ou on ne peut pas) toutes les énumérer.

9.1 Améliorer la valeur de relaxation

Afin de réduire le nombre de branchement on ajoute des contraintes pour obtenir une meilleure valeur de relaxation (coût de la solution du PL relaxé). En effet dans le *branch and bound* si la solution courante est fractionnaire on branche ; dans le *branch and cut* au lieu de brancher directement on cherche une nouvelle contrainte (appelée *coupe*) qui élimine cette solution tout en conservant toutes les solutions entières. On

répète ce procédé itérativement jusqu'à ce que les nouvelles coupes générées ne changent plus beaucoup la valeur de relaxation. Si la solution obtenue n'est pas entière on branche et on recommence le même processus sur les fils. Le fait d'ajouter des contraintes (ou coupes) pour améliorer la valeur de relaxation réduit le nombre de branchements à effectuer, on construit donc un arbre plus petit (moins de mémoire nécessaire, parcours de l'arbre plus rapide).

Pour mieux comprendre ce processus considérons l'exemple de la partie 2 :

$$(P)\begin{cases} \text{Minimiser} & -8x_0 - 5x_1 \\ \text{sous} & \\ & x_0 + x_1 \leq 6 \\ & 9x_0 + 5x_1 \leq 45 \\ & x_0, x_1 \in \mathbb{N} \end{cases}$$

On a vu comment résoudre ce problème par *branch and bound* dans la partie 4.3. Cette résolution implique la création de 6 sous-problèmes (et donc la résolution de 7 PL en comptant le PL de la racine) pour un problème de 2 variables et 2 contraintes. Pour résoudre ce problème en nombres entiers SCIP ajoute la coupe $3x_0 + 2x_1 \leq 15$ après avoir résolu le problème relaxé : la solution fractionnaire est $(x_0 = 3.75\,;\, x_1 = 2.25)$. On voit sur la figure 42 que cette contrainte coupe le point fractionnaire et conserve les solutions entières. La résolution du nouveau PL relaxé donne $(x_0 = 5\,;\, x_1 = 0)$ qui est entier, on n'a pas besoin de brancher.

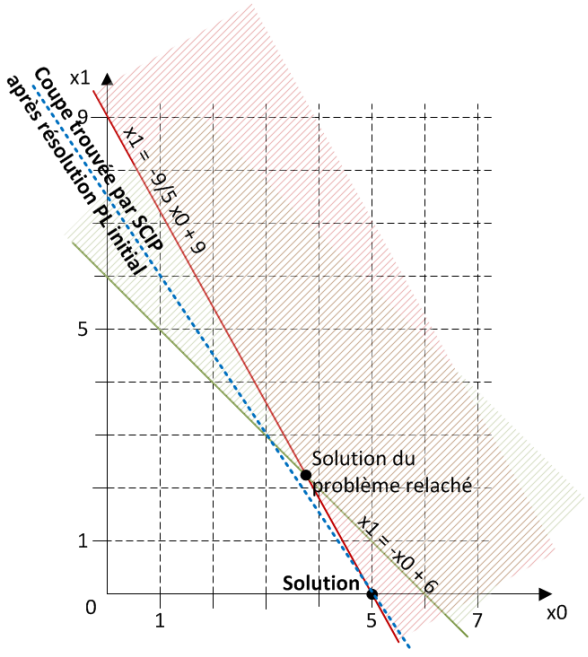

FIGURE 42 – Ajout d'une coupe (en bleu)

Remarques : Par défaut, les solveurs linéaires utilisent des coupes générales pour limiter au maximum le nombre de branchements : parmi les plus connues on peut citer les coupes de Gomory.

9.2 Eviter d'énumérer un nombre exponentiel de contraintes

Considérons le problème du voyageur de commerce (TSP, *Traveller Salesman Problem*) : Etant donné un graphe non orienté $G = (V, E)$ dans lequel les arcs $e \in E$ ont un coût c_e on cherche un cycle hamiltonien C (i.e. un cycle qui passe par tous les sommets une et une seule fois) de coût minimal. Un formulation PLNE possible est la suivante :

$$(TSP)\begin{cases} \text{Minimiser} & \sum_{e \in E} c_e x_e & & \\ \text{sous} & & & \\ & \forall v \in V, & \sum_{e \in \delta(v)} x_e = 2 & (1) \\ & \forall W \subsetneq V, W \neq \emptyset, & \sum_{e \in \delta(W)} x_e \geq 2 & (2) \\ & \forall e \in E, & x_e \in \{0, 1\} & \end{cases}$$

où x_e est la variable binaire qui vaut 1 si l'arc e est dans C, 0 sinon. Les contraintes sont :

(1) contraintes de degré : chaque sommet est impliqué dans deux arcs de C (on arrive et on repart).

(2) contraintes de coupe : il faut au moins un arc qui entre et un arc qui sort de chaque sous-tour de C. Autrement dit, chaque ensemble non vide de sommets $W \subsetneq V$ doit être relié à $V - W$ par au moins 2 arcs, cela oblige C à être connexe.

Il est clair que les contraintes (2) sont en nombre exponentiel. L'idée est donc de résoudre ce problème par *branch and cut* : on limite le PL initial aux contraintes de type (1) et on le résout en fractionnaire. On cherche si la solution viole une contrainte de type (2). Pour cela il suffit de chercher une coupe minimale (problème de la *coupe-min*) dans le graphe G en considérant que les arcs de G sont valués par les x_e : si la coupe minimale respecte la contrainte (2) alors toutes les coupes la respectent aussi ; sinon on ajoute une contrainte de type (2) en prenant $W =$ la coupe minimale. De cette manière on évite l'énumération de toutes les coupes. Une fois toutes les coupes nécessaires ajoutées on branche et on recommence ce processus dans les fils.

9.3 Implémentation efficace

Comme on l'a vu dans la partie 7.3 concernant l'implémentation efficace d'un *branch and price* il ne faut pas relancer la résolution du PL après chaque ajout de contrainte : le simplexe n'ayant en général que très peu d'itérations à faire pour trouver la nouvelle solution à partir de l'ancienne. Il faut donc conserver les données du simplexe pour

reprendre rapidement la résolution. C'est pourquoi il est commode d'utiliser une bibliothèque comme **SCIP** pour programmer ce genre de méthode (voir section suivante).

10 Programmer un *branch and cut* avec SCIP

Dans cette section on présente le principe général pour programmer un *branch and cut* avec **SCIP** : on présente le module, ses principales méthodes et la manière dont elles interagissent. Par contre on ne présente pas tous les détails d'implémentation ni la signification de tous les paramètres (voir le site de **SCIP** [1] pour cela).

10.1 Le *Constraint Handler* : présentation

Le module de **SCIP** qui gère l'ajout de contraintes dynamiques s'appelle le *Constraint Handler*. Il est implémenté par la classe ObjConshdlr. Il permet de gérer différents types de contraintes dynamiquement : les coupes, les contraintes non linéaires ou encore les décisions de branchement (voir section 8.3.2). La classe ObjConshdlr est abstraite. Dans le cadre du *branch and cut* les méthodes que l'on va utiliser sont : scip_check, scip_enfolp, scip_enfops, scip_sepalp, scip_sepaps qui sont toutes virtuelles pures. Un *Constraint Handler* ne peut être utilisé que pour un seul type de contraintes. Si on a différents types de contraintes il faut alors plusieurs *Constraint Handlers*.

10.2 *Constraint Handler* vs *Separator*

Il existe un autre module de **SCIP** qui permet d'ajouter des contraintes dynamiquement : le *separator*. Le *separator* est utilisé plutôt pour ajouter des coupes qui servent à améliorer la valeur de relaxation (voir 9.1) mais dont on peut se passer du point de vue de la faisabilité de la solution. De plus ces coupes doivent être exprimées sous forme de contraintes linéaires.

Le *constraint handler* peut implémenter des contraintes non linéaires et des contraintes liées à des décisions de branchement. On l'utilise aussi pour les contraintes linéaires mais trop nombreuses pour être toutes énumérées (voir 9.2). Il rentre dans un cadre d'utilisation beaucoup plus large que le *Separator* et est plus complexe à utiliser. C'est pourquoi on a choisi dans ce document de présenter le *constraint handler* plutôt que le *separator*.

10.3 Le *Constraint Handler* : utilisation

Les exemples de code de cette partie sont extraits de l'exemple du TSP disponible sur le site de **SCIP** ([1]).

10.3.1 Dériver la classe ObjConshdlr et informer SCIP

Comme pour tous les autres modules la première étape est de créer une classe fille à ObjConshdlr et l'inclure dans le processus de résolution de **SCIP** (voir figure 43). Le

constraint handler de la figure 43 est inclus dans **SCIP** par la commande SCIPincludeObjConshdlr(scip, new ConshdlrSubtour(scip), TRUE) placée avant l'appel à SCIPsolve().

```
/** C++ constraint handler for TSP subtour elimination constraints */
class ConshdlrSubtour : public scip::ObjConshdlr
{

public:
   /** default constructor */
   ConshdlrSubtour( SCIP* scip ): ObjConshdlr(scip, "subtour",
     "TSP subtour elimination", 1000000, -2000000, -2000000,
     1, -1, 1, 0, FALSE, FALSE, FALSE, TRUE, SCIP_PROPTIMING_BEFORELP)
   {   }
```

FIGURE 43 – Dériver ObjConshdlr

10.3.2 Les méthodes principales

Il y a cinq méthodes principales :

scip_check() qui sert à vérifier si la solution courante respecte toutes les contraintes du *Constraint Handler* ;

scip_sepalp() qui sert à implémenter les coupes dont le but est d'améliorer la valeur de relaxation. Cette méthode n'est appelée que sur la solution optimale du LP courant ;

scip_enfolp() qui est appelée après que scip_sepalp ait ajoutée toutes ses coupes, sur une solution optimale pour le PL courant et si la solution n'est pas faisable. Elle doit alors résoudre l'infaisabilité soit en ajoutant une contrainte, soit en réduisant le domaine de certaines variables, soit en branchant ;

scip_sepaps() a le même rôle que scip_sepalp mais est appelée dans le cas où on n'a pas pu résoudre le PL courant à optimalité (par choix de l'utilisateur ou à cause de difficultés numériques). On travaille donc sur une solution du PL courant non optimale ;

scip_enfops() a le même rôle que scip_enfolp mais est appelée dans le cas où on n'a pas pu résoudre le PL courant à optimalité (par choix de l'utilisateur ou à cause de difficultés numériques). On travaille donc sur une solution du PL courant non optimale.

Il y a une sixième méthode non virtuelle pure mais importante c'est scip_lock(). Pour chaque contrainte elle doit informer **SCIP** sur la manière dont les variables lui appartenant peuvent être arrondies sans que la contrainte soit violée. Pour cela elle utilise la fonction SCIPaddVarLocks().

La figure 44 donne le schéma général de la phase de *cutting* d'un nœud dans le cas où on travaille sur des solutions du PL courant optimales. On donne le schéma pour un *cutting* classique : l'ordre des différentes étapes peut être modifié à l'aide des paramètres attributs de la classe ObjConshdlr.

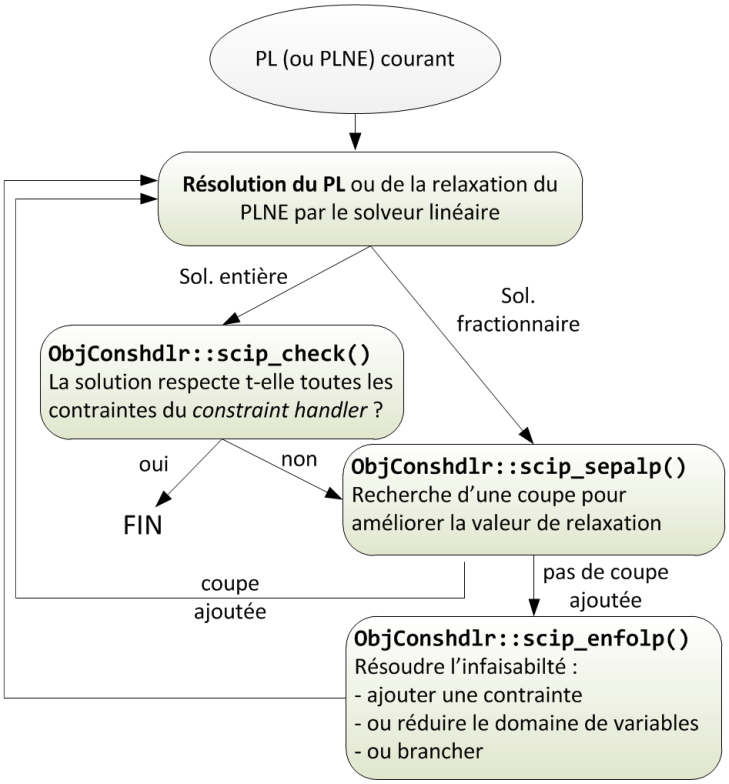

FIGURE 44 – Schéma général d'une phase de *cutting* classique

11 Synthèse : fonctionnement du *branch cut and price*

Les sections précédentes nous ont permis de comprendre le fonctionnement des 3 méthodes qui composent le *branch cut and price* : le *branch and bound*, le *branch and cut* et le *branch and price*. La figure 45 montre la manière dont ces méthodes s'articulent dans le cadre d'un *branch cut and price* ainsi que les classes de **SCIP** impliquées dans la résolution.

Toute la difficulté de programmation d'un *branch cut and price* est justement de faire s'articuler correctement les différentes étapes de résolution. D'autre difficultés théoriques viennent s'y ajouter :

– des problèmes de convergence : les coupes (étape de *cutting*) tendent à faire augmenter (cas d'une *minimisation*) le coût de la solution puisqu'on ajoute des contraintes tandis que la génération de variables (étape de *pricing*) tend à faire diminuer le coût de la solution puisqu'on ajoute des variables améliorantes. Ces deux étapes ont donc tendance à s'annuler l'une l'autre. On peut donc être amené à générer un très grand nombre de variables et de coupes avant d'avoir une solution stable sur laquelle on peut brancher.

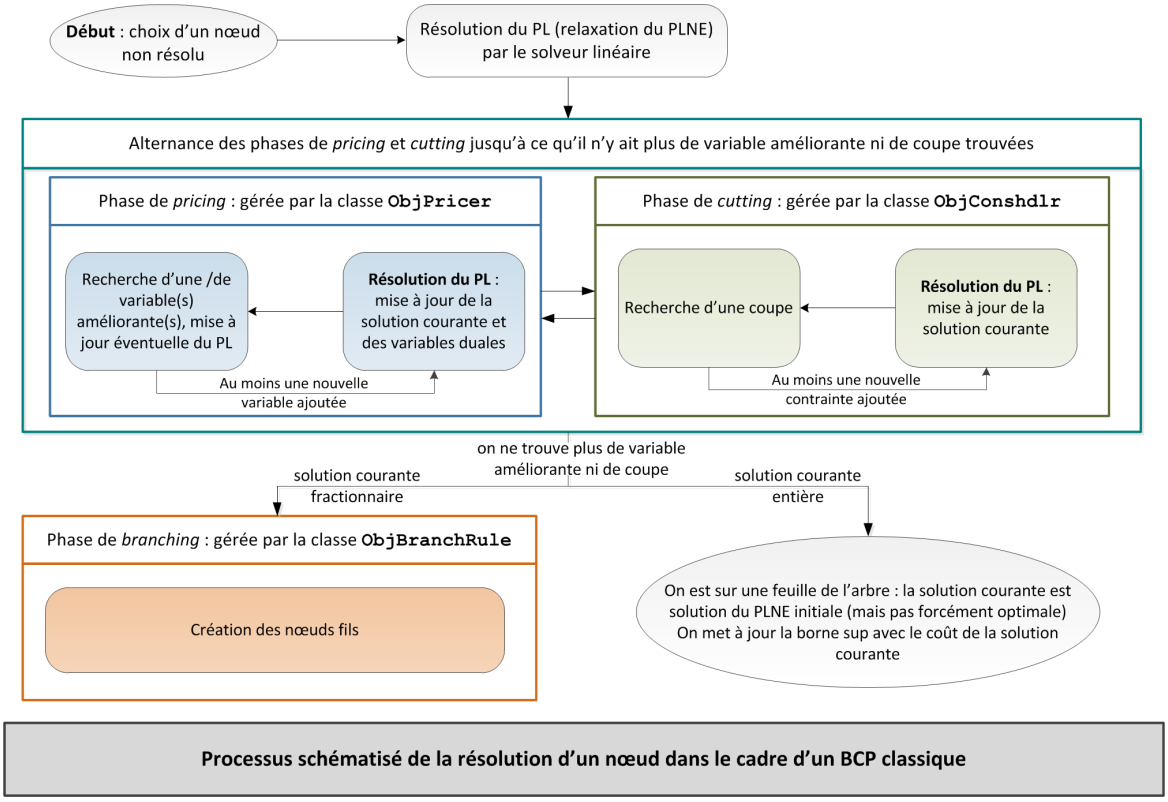

FIGURE 45 – Résolution d'un nœud dans le cadre d'un BCP

– des problèmes liés au branchement : trouver une bonne règle de branchement (qui coupe la solution courante sans compromettre le *pricing* et qui divise l'espace des solutions de manière équilibrée) est indispensable pour la réussite d'un *branch cut and price* sinon on risque de ne jamais s'arrêter de brancher.

– des problèmes liés à la formulation du PL : les problèmes en nombres entiers uniquement ou réels uniquement posent moins de difficultés du point de vue de la convergence que les problèmes mixtes. En effet dans les problèmes mixtes l'expérience prouve que les contraintes de couplage tendent généralement à freiner la convergence des variables entières vers des valeurs entières.

12 Conclusion

Les différentes parties de ce document donnent les clés pour comprendre le fonctionnement d'un *branch cut and price* et la manière de le programmer efficacement à l'aide de la bibliothèque **SCIP**. A ma connaissance les publications qui font état d'un *branch cut and price* travaillent sur un PLNE (et non sur un PL mixte) et utilisent principalement l'étape de *cutting* pour améliorer la valeur de la relaxation mais rarement pour

générer des contraintes indispensables à la faisabilité du problème, ce dernier point étant très délicat à mettre en œuvre à cause des problèmes de convergence qu'il implique.

Références

[1] http ://scip.zib.de/

[2] Programmation linéaire en nombres entiers -Résolution : coupes et séparation - évaluation, Hugues Talbot, Laboratoire A2SI, 2009.

[3] Jacques Teghem. Programmation linéaire. Editions de l'université de Bruxelles, Editions Ellipses. 2004

[4] G. Fleury, P. Lacomme. Programmation linéaire avancée. Ellipses. 2009.

[5] T.K. Ralphs, L. Ladányi. COIN/BCP User's Manual. 2001.

[6] A. Mehrotra, M. A. Trick. A column generation approach for graph coloring. Informs. vol 8, no 4. 1996.