Programmer un *branch* & *price* en C++ avec la librairie SCIP 4.0

Hélène Toussaint, janvier 2018

But : apprendre à programmer un branch & price avec la librairie SCIP 4.0 en C++.

Prérequis : Ce document est la suite de [4] qui présente SCIP et les techniques de *branch cut & price*. Pour aborder ce document il est nécessaire de :

- avoir une bonne connaissance du C++,
- avoir de bonnes notions de programmation linéaire (savoir écrire un modèle linéaire, et avoir des notions sur la dualité),
- savoir utiliser SCIP pour résoudre un programme linéaire simple (voir [4]).

Table des matières

1 Introduction

Le but de ce document est d'apprendre à programmer un *branch* & *price* en C++ à l'aide du *framework* SCIP (version 4.0).

La première partie est consacrée au *branch* & *price* :

- Rappels théoriques de la méthode (sections 2 et 3)
- Application au problème de coloration de graphe (sections 4 et 5)

La seconde partie est consacrée à la programmation avec SCIP :

- Présentation du *framework* SCIP, description des classes et méthodes utilisées dans le cadre d'un *branch* & *price* (section 6)
- Exemple de programme complet pour résoudre le problème de coloration de graphe (section 7 et programme à télécharger [6])
- Présentation des améliorations possibles pour rendre le programme plus efficace.

Remarque : ce document ne constitue pas un cours sur le *branch* & *price*. Les rappels théoriques sont très rapides et visent seulement à présenter les connaissances théoriques minimales à avoir pour être capable de résoudre un programme linéaire en nombres entiers par *branch* & *price*.

Partie 1

Branch & *price* : rappels théoriques et application au problème de coloration de graphe

2 Notations et vocabulaire

Cette section présente une synthèse des notations générales et du vocabulaire utilisé dans la suite du document.

2.1 Notations

On considère dans cette première partie un programme linéaire formé de n variables fractionnaires et m contraintes qui s'écrit sous la forme suivante :

$$
(\mathcal{P}_{init}) \quad
\begin{aligned}
&\text{Minimiser } \sum_{i=1}^{n} c_i x_i \\
&\text{sous} \\
&\forall j \in \{1, \dots, m\}, \quad \sum_{i=1}^{n} a_{ji} x_i \geq d_j \\
&\forall i \in \{1, \dots, n\}, \quad x_i \in \mathbb{R}^+
\end{aligned}
$$

On notera de plus :

- $A \in M_{m,n}(\mathbb{R})$ $(A = (a_{ji})_{j=1,..,m;i=1,..,n})$: matrice des contraintes
- $\forall i \in \{1, \dots, n\}, (a_{ji})_{j=1..m}$: colonne de la matrice des contraintes associée à la variable i
- $\forall j = \{1, \dots m\}, \lambda_j$: variable duale associée à la $j^{\text{ème}}$ contrainte
- $\forall j = \{1, \dots m\}, \mu_j$: multiplicateur de Farkas associé à la $j^{\text{ème}}$ contrainte

Remarque : ici l'indice i est associé aux variables et donc aux colonnes de la matrice tandis que l'indice j est associé aux contraintes et donc aux lignes de la matrice.

2.2 Vocabulaire et abréviations

2.2.1 Vocabulaire général

- **PL** : programme linéaire,
- **PLNE** : programme linéaire en nombres entiers,
- une **variable** est associée à une colonne de la matrice des contraintes, on utilise dans la suite indifféremment variable ou colonne.

2.2.2 Vocabulaire du *branch* & *price*

- **problème initial** : le problème linéaire initial $(\mathcal{P})$ (avec toutes les variables) que l'on souhaite résoudre par *branch* & *price*,
- **problème maître restreint** (**PMR**) ou **problème maître** : le problème linéaire contenant toutes les contraintes de $(\mathcal{P})$ mais seulement une partie des variables,
- ***pricing*** : étape de la génération de colonne consistant à rechercher une variable améliorante.

3 Rappels sur la génération de colonnes, le *branch* & *bound* et le *branch* & *price*

Cette section présente les notions théoriques minimales à connaître pour pouvoir programmer un *branch* & *price*. Le *branch* & *price* repose sur la technique de génération de colonnes. On rappelle donc dans cette section :

1. comment se déroule la résolution d'un programme linéaire (PL) par génération de colonnes (section 3.1)
2. comment la génération de colonnes est utilisée dans un *branch* & *price* (section 3.2).

Chacune des étapes de ces méthodes sera illustrée dans la partie 5 sur le problème de coloration de graphe.

3.1 La génération de colonnes

La génération de colonnes est une méthode permettant de résoudre de manière exacte des problèmes linéaires en nombres fractionnaires. Elle est généralement utilisée pour des problèmes linéaires contenant un très grand nombre de variables (tellement grand qu'il est parfois impossible d'énumérer toutes les variables dans un temps "raisonnable").

L'idée de la génération de colonnes repose sur le fait qu'un PL peut être résolu sans énumérer explicitement l'ensemble de ses variables (ce résultat s'appuie sur la théorie de la dualité, voir [1]). Le principe de la génération de colonnes est alors de générer "certaines" variables du PL jusqu'à ce les variables générées soient suffisantes pour exprimer la solution optimale.

On distingue 2 étapes dans la génération de colonnes :

1. la recherche de variables initiales (génération d'un "petit" sous-ensemble de variables)
2. la recherche de variables améliorantes (cette étape s'appelle *pricing*).

3.1.1 Recherche de variables initiales : construction du Problème Maître Restreint (PMR)

Notons $(\mathcal{P})$ le PL que l'on souhaite résoudre par génération de colonnes.

L'étape d'initialisation consiste à sélectionner un sous-ensemble des variables parmi l'ensemble des variables de $(\mathcal{P})$.

Pour cela il suffit, par exemple, d'expliciter une solution de $(\mathcal{P})$ (solution triviale ou calculée à l'aide d'une heuristique) et d'utiliser comme ensemble de variables initiales les variables composant cette solution (voir exemple section 5.2).

Le PL $(\mathcal{P}_M)$ formé du sous-ensemble de variables et de l'ensemble des contraintes $(\mathcal{P})$ est appelé *problème maitre restreint (PMR)* ou plus simplement *problème maître*.

Ce sous-ensemble de variables est évidemment souvent insuffisant pour exprimer la solution optimale. La prochaine étape consiste donc à générer itérativement de nouvelles variables jusqu'à obtenir la certitude (via la dualité) que toutes les variables qui forment la solution optimale ont été générées.

3.1.2 Recherche de variables améliorantes (*pricing*)

Principe

La recherche de nouvelles variables est une méthode itérative et s'appuie sur la dualité : ce sont les variables duales qui permettent de calculer la (ou les) prochaines variables *améliorantes*. On parle de variables *améliorantes* car elles sont susceptibles d'améliorer le coût de la fonction objectif si on les ajoute au PMR (*problème maitre restreint*).

A chaque itération on tente de générer une / des nouvelles variables, une itération comporte les étapes suivantes :

1. résoudre le problème maître restreint $(\mathcal{P}_M)$ (à l'aide d'un solveur linéaire),
2. récupérer la solution duale de $(\mathcal{P}_M)$ (fournie par le solveur),
3. utiliser cette solution duale pour trouver (via la résolution d'un problème auxiliaire) une (ou plusieurs) variables améliorantes,
4. ajouter les nouvelles variables à $(\mathcal{P}_M)$.

Les itérations s'arrêtent quand il n'existe plus de variable améliorante, la solution du problème maître restreint $(\mathcal{P}_M)$ est alors la solution optimale de $(\mathcal{P})$.

L'étape délicate est l'étape 3 qui consiste à créer et résoudre un problème auxiliaire (appelé *sous-problème*) pour générer une nouvelle variable. Elle est détaillée ci-dessous.

Création et résolution du sous-problème

Le sous-problème est le problème à résoudre pour trouver une variable améliorante.

Les résultats sur la dualité (voir [1] ou [2]) permettent d'affirmer que, pour un problème de minimisation, une variable est améliorante si elle a un coût réduit strictement négatif. Supposons que le PL maître restreint possède m contraintes, le coût réduit $\bar{c}$ d'une variable se calcule alors de la manière suivante :

$$\bar{c} = c - \sum_{j=1}^{m} \lambda_j \, a_j$$

où

- c est le coefficient de la variable dans la fonction objectif
- λ_j est la variable duale associée à la contrainte j $(j = 1, \dots, m)$
- a_j est le coefficient de la variable dans la contrainte j
- m est le nombre de contraintes

Le sous-problème consiste donc à chercher les coefficients $(a_j)_{j=1,\dots m}$ de la nouvelle colonne tels que : $c - \sum_{j=1}^{m} \lambda_j \, a_j < 0$.
Autrement dit, on cherche les coefficients $(a_j)_{j=1,\dots m}$ de la nouvelle colonne tels que : $\sum_{j=1}^{m} \lambda_j \, a_j > c$.

Le sous problème à résoudre est donc le suivant :

(SsPb) : "*trouver une colonne C =* $(a_j)_{j=1,\dots m}$ *telle que* $\sum_{j=1}^{m} \lambda_j \, a_j > c$ *et C représente une variable du problème initial* $(\mathcal{P})$*.*"

Ce sous-problème peut être résolu soit à l'aide d'un solveur linéaire (s'il peut être modélisé sous forme de PL ou PLNE, ce qui est généralement le cas), soit à l'aide d'un algorithme *ad hoc.*

Si (SsPb) n'a pas de solution alors la génération de colonnes s'arrête : les colonnes constituant la solution optimales ont déjà été toutes générées et la solution actuelle du PMR est donc la solution du problème initial $(\mathcal{P})$.

Voir l'exemple section 5.3.

Génération de variables pour la réalisabilité (cas PL maître non réalisable)

Il peut arriver que le problème maître restreint (PMR) soit non faisable (il ne possède pas de solution). Cela peut signifier 2 choses :

1. le problème initial $(\mathcal{P})$ n'a pas de solution, donc le PMR non plus,
2. le problème initial $(\mathcal{P})$ a une solution mais le sous ensemble de variables choisis durant la phase d'initialisation est insuffisant pour que le PMR ait une solution.

Il faut alors essayer de générer des nouvelles variables qui tendent à rendre le PMR réalisable. Pour cela on peut s'appuyer sur le lemme de Farkas dont la version matricielle s'énonce de la manière suivante :

Soit A une matrice de $\mathcal{M}_{mn}(\mathbb{R})$, et $b \in \mathbb{R}^m$, alors :
Soit le système $Ax = b, x \geq 0$ admet une solution, **soit** il existe $\mu \in \mathbb{R}^m$ tel que $\mu^T A \leq \mathbf{0}$ *et* $\mu^T b > 0$

Remarque : $\mathbf{0}$ est le vecteur nul, $\mu^T A \leq \mathbf{0}$ signifie que chaque composante de $\mu^T A$ est inférieur ou égal à zéro.

La démonstration de ce lemme s'appuie sur les théorèmes de dualité (voir [1] page 103).

Les éléments du vecteur μ sont appelés *multiplicateurs de Farkas*.

Lorsqu'un PL n'est pas réalisable les solveurs (comme CPLEX ou SCIP par exemple) fournissent les multiplicateurs de Farkas. On peut alors les utiliser pour générer une nouvelle variable qui tend à rendre le PL réalisable. Or, d'après le lemme, le système $Ax = b, x \geq 0$ admet une solution si ($\mu^T A \leq \mathbf{0}$ *et* $\mu^T b > 0$) est faux. Pour rendre le système réalisable il suffit donc de générer une nouvelle colonne $\boldsymbol{a} = (a_j)_{j=1,\dots m}$ de A telle que $\mu^T \boldsymbol{a} > 0$.

Autrement dit on cherche les coefficients $(a_j), j = 1, \dots, m$ de la nouvelle colonne tels que : $\sum_{j=1}^{m} \mu_j a_j > 0$.

Dans ce cas le sous problème à résoudre est :

(SsPb2) : "*trouver une colonne* $\boldsymbol{a} = (a_j)_{j=1,\dots m}$ *telle que* $\sum_{j=1}^{m} \mu_j a_j > 0$ *et* $\boldsymbol{a}$ *représente une variable du problème initial* $(\mathcal{P})$*.*"

Remarque : dans le cas où le problème maître restreint est non faisable les variables duales n'existent pas (en effet, il n'existe ni solution primale ni solution duale).

3.1.3 Synthèse sur la génération de colonnes

La démarche pour résoudre un programme linéaire fractionnaire $(\mathcal{P})$ par génération de colonnes est la suivante :

1. on calcule un sous-ensemble $\mathcal{V}$ des variables de $(\mathcal{P})$,
2. on forme un programme linéaire $(\mathcal{P}_M)$, appelé problème maître restreint, constitué des contraintes de $(\mathcal{P})$ est du sous-ensemble de variables $\mathcal{V}$,
3. on ajoute des variables à $(\mathcal{P}_M)$ de manière itérative. La recherche de nouvelles variables s'appelle *pricing*. Elle consiste à chercher une nouvelle colonne $(a_j)_{j=1..m}$ (de la matrice des contraintes), qui représente une variable de $(\mathcal{P})$, et qui vérifie l'une des deux équations suivantes (suivant que le PMR est réalisable ou non) :

$\sum_{j=1}^{m} \lambda_j a_j > c$	$\sum_{j=1}^{m} \mu_j a_j > 0$
Cas PL réalisable : utilisation des variables duales $\lambda = (\lambda_j)_{j=1..m}$ et du coefficient objectif c de la nouvelle variable	**Cas PL non réalisable** : utilisation des multiplicateurs de Farkas $\mu = (\mu_j)_{j=1..m}$

4. Si le pricing ne donne pas de nouvelle variable alors la résolution s'arrête : la solution optimale de $(\mathcal{P}_M)$ est la solution optimale de $(\mathcal{P})$.

Remarque : tous les PL ne sont pas adaptées à la génération de colonnes (un même problème peut avoir plusieurs formulations linéaires, certaines adaptées à la génération de colonnes, d'autres non).

La Figure 1 montre l'enchainement des différentes étapes d'une résolution par génération de colonnes.

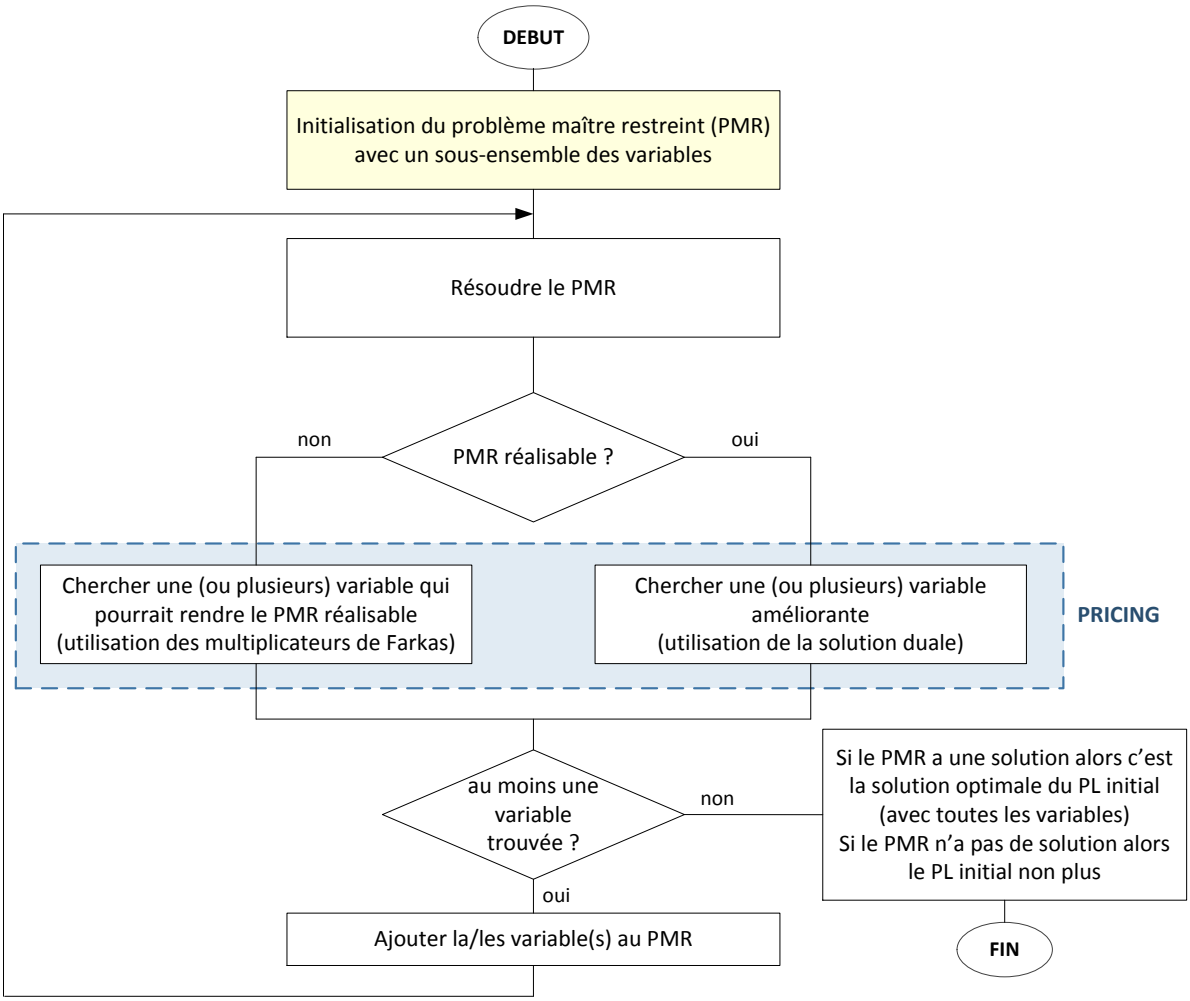


Figure 1. Schéma général de résolution d'un PL par génération de colonnes

3.2 Le *branch* & *bound*

Le *branch & bound* est une technique utilisée pour résoudre des problèmes d'optimisation combinatoire, et notamment des problèmes modélisés par un programme linéaire en nombres entiers (PLNE). Il est indispensable de bien comprendre comment fonctionne un *branch* & *bound* pour comprendre comment fonctionne un *branch & price*.

Un *branch* & *bound* (B&B), ou algorithme par séparation et évaluation, est un algorithme fréquemment utilisé pour résoudre des problèmes d'optimisation combinatoire. La difficulté dans les problèmes d'optimisation combinatoire vient du fait que l'ensemble des solutions du problème peut être très grand si bien qu'il est impossible d'énumérer toutes les solutions pour trouver la meilleure.

Le *branch* & *bound* est constitué de 2 phases : la séparation (*branch*) et l'évaluation (*bound*).
L'idée de la phase de *séparation* est de séparer le problème principal en sous-problèmes, de sorte que chaque sous-problème soit "plus simple" à résoudre ("plus simple" dans le sens où il est associé à un ensemble de solutions plus petit que le problème initial).
La qualité de chaque sous-problème est ensuite évaluée à l'aide d'un algorithme *ad hoc,* c'est la phase d'*évaluation* (qui permet de déduire une borne au problème initial).

Le *branch* & *bound* est une technique de résolution qui ne nécessite donc pas forcément l'expression du problème sous forme linéaire. Néanmoins, dans les sections suivantes on décrit son fonctionnement dans le cadre de la résolution d'un PLNE (problème linéaire en nombres entiers).

3.2.1 La séparation

Soit $(\mathcal{P})$ le problème linéaire en nombres entiers (PLNE) que l'on souhaite résoudre et $\mathcal{S}$ l'ensemble de toutes les solutions réalisables de $(\mathcal{P})$.

L'étape de séparation consiste à créer 2 sous problèmes $(\mathcal{P}_1)$ et $(\mathcal{P}_2)$ à partir de $(\mathcal{P})$. $(\mathcal{P}_1)$ et $(\mathcal{P}_2)$ possèdent toutes les contraintes de $(\mathcal{P})$ ainsi qu'une contrainte additionnelle qui est propre à chaque sous-problème.

L'ajout d'une contrainte additionnelle réduit naturellement l'espace des solutions. On divise ainsi l'espace des solutions $\mathcal{S}$ en 2 sous espaces $\mathcal{S}_1$ et $\mathcal{S}_2$ de taille plus petite. Bien évidemment, il est nécessaire que $\mathcal{S}_1$ et $\mathcal{S}_2$ forment un recouvrement de l'ensemble des solutions initial $\mathcal{S}$ (et idéalement une partition).

Chaque sous-problème peut à son tour être divisé en sous-problèmes. La séparation récursive des sous-problèmes fait alors apparaître un arbre dont chaque nœud est un sous-problème contenant toutes les contraintes du problème père auxquelles s'ajoutent une (ou plusieurs) nouvelles contraintes.

L'arbre ainsi construit est appelé ***arbre de recherche*** et les contraintes additionnelles sont appelées ***contraintes de branchement***.

Pour que la méthode soit efficace il faut éviter de générer un trop grand nombre de sous-problèmes. Pour cela il faut :

1. choisir judicieusement les contraintes de branchement (et il est souvent très difficile de savoir a priori quelles sont les contraintes les plus judicieuses...)
2. évaluer la "qualité" des sous-problèmes créés (c'est la phase d'*évaluation,* voir section suivante).

Remarque : Par abus de langage on dira "évaluation d'un nœud" pour parler de l'évaluation du problème associé au nœud.

3.2.2 L'évaluation

Afin d'avoir un *branch & bound* efficace, il est nécessaire d'avoir un algorithme qui permet d'évaluer "rapidement" la qualité d'un sous-problème. Autrement dit on cherche une estimation (optimiste) de la meilleure solution qu'il est possible de trouver dans le sous-ensemble de solutions liées au sous-problème considéré. Cela permet de :

1. traiter en priorité les sous-problèmes qui semblent les plus prometteurs
2. éviter de résoudre des sous-problèmes qui ne peuvent pas contenir la solution optimale

Dans le cadre d'un problème de minimisation on a donc besoin d'un algorithme qui fournisse une borne inférieure au problème. Une manière très classique d'obtenir une borne inférieure d'un PLNE est de résoudre sa relaxation fractionnaire.

Remarque : le *branch & bound* sera d'autant plus efficace si on dispose de bornes supérieures de bonne qualité. Pour cela on utilise souvent des heuristiques.

3.2.3 *Branch* & *bound* : l'algorithme

Les différentes étapes d'un *branch* & *bound* sont les suivantes (dans le cadre de la minimisation d'un programme linéaire en nombres entiers (PLNE)) :

1. Initialisation : le PLNE initial devient le PLNE courant (que l'on va évaluer)

2. **Phase d'évaluation** : on évalue le PLNE courant (en résolvant sa relaxation fractionnaire), on obtient une solution fractionnaire S dont le coût est une borne inférieure au problème courant (supérieure si on traite un problème de maximisation)

3. **Phase de séparation** : Si la solution S est fractionnaire alors on crée au moins 2 sous-problèmes. Une manière classique de former les sous-problèmes est de choisir une variable x qui a un coût fractionnaire α dans la solution S. La contrainte additionnelle est alors $x \leq \lfloor \alpha \rfloor$ pour le premier sous-problème et $x \geq \lceil \alpha \rceil$ pour le second. On remarque que l'ensemble des solutions entières est préservé et que la solution S (dans laquelle $x = \alpha$) n'est plus solution dans aucun des sous-problèmes. Les sous-problèmes sont stockés en mémoire.
   Si la solution obtenue en 2. est entière alors aucun sous-problème n'est créé.

4. Choix du prochain problème (PLNE) à évaluer : parmi les sous-problèmes stockés en mémoire et non encore évalués on en choisit un qui devient le problème courant. On retourne en 2.

L'algorithme se termine quand tous les sous-problèmes ont été évalués. La solution optimale est alors la meilleure solution entière obtenue en phase 2.

3.2.4 Exemple

La Figure 2 montre un arbre de recherche lié à la résolution du PLNE $(\mathcal{P}_{ex})$ suivant par *branch & bound* :

$(\mathcal{P}_{ex})$

$$\begin{aligned} &\text{Minimiser } \mathbf{obj} : 3x - 5y \\ &\text{Sous} \\ &\quad 2x + \ y \geq 7 \\ &\quad x - 6y \geq 1 \\ &\quad x, y \in \mathbb{N} \end{aligned}$$

Le déroulement du *branch* & *bound* est le suivant :

- $(\mathcal{P}_{ex})$ est évalué (i.e. on résout sa relaxation fractionnaire) : on obtient la solution ($x = 3.308$; $y = 0.385$) de coût 8

- On choisit de créer deux sous-problèmes $(\mathcal{P}_{ex})_2$ et $(\mathcal{P}_{ex})_3$ (phase de séparation) en *branchant* sur la variable x : on associe donc respectivement à $(\mathcal{P}_{ex})_2$ et $(\mathcal{P}_{ex})_3$ les contraintes

additionnelles $x \leq 3$ et $x \geq 4$. (Remarque : ici le choix de x est arbitraire, on aurait pu choisir y).

- On évalue $(\mathcal{P}_{ex})_2$, il n'a pas de solution, il n'y a donc pas de phase de séparation à partir de ce problème.

- On évalue $(\mathcal{P}_{ex})_3$, la solution est $(x = 4 ; y = 0.5)$ de coût 9.5

- On crée deux sous-problèmes $(\mathcal{P}_{ex})_4$ et $(\mathcal{P}_{ex})_5$ (phase de séparation) en *branchant* sur la variable y : on associe donc respectivement à $(\mathcal{P}_{ex})_4$ et $(\mathcal{P}_{ex})_5$ les contraintes additionnelles $y \leq 0$ et $y \geq 1$. (Remarque : ici seule la variable y peut être utilisée pour le branchement, la variable x étant entière).

- On évalue $(\mathcal{P}_{ex})_4$, on obtient une solution entière de coût 12, la solution étant entière il n'y a pas de phase de séparation

- On évalue $(\mathcal{P}_{ex})_5$, on obtient une solution entière de coût 16, la solution étant entière il n'y a pas de phase de séparation

- Tous les sous-problèmes crées ont été évalués : la meilleure solution entière est donc la solution optimale. Il s'agit de $(x = 4 ; y = 0)$ de coût 12.

Remarque 1 : la forme de l'arbre de recherche dépend des décisions de branchement qui ont été prises.

Remarque 2 : la plupart des solveurs (CPLEX par exemple) utilisent un *branch* & *bound* pour trouver la solution d'un PLNE mais c'est complétement transparent pour l'utilisateur car tout est fait automatiquement par le solveur. Dans le cadre du *branch* & *price* par contre, c'est différent. L'utilisateur doit intervenir notamment au moment de la résolution d'un nœud et au moment de la séparation. C'est pourquoi il est nécessaire d'avoir une bonne compréhension de cette méthode.

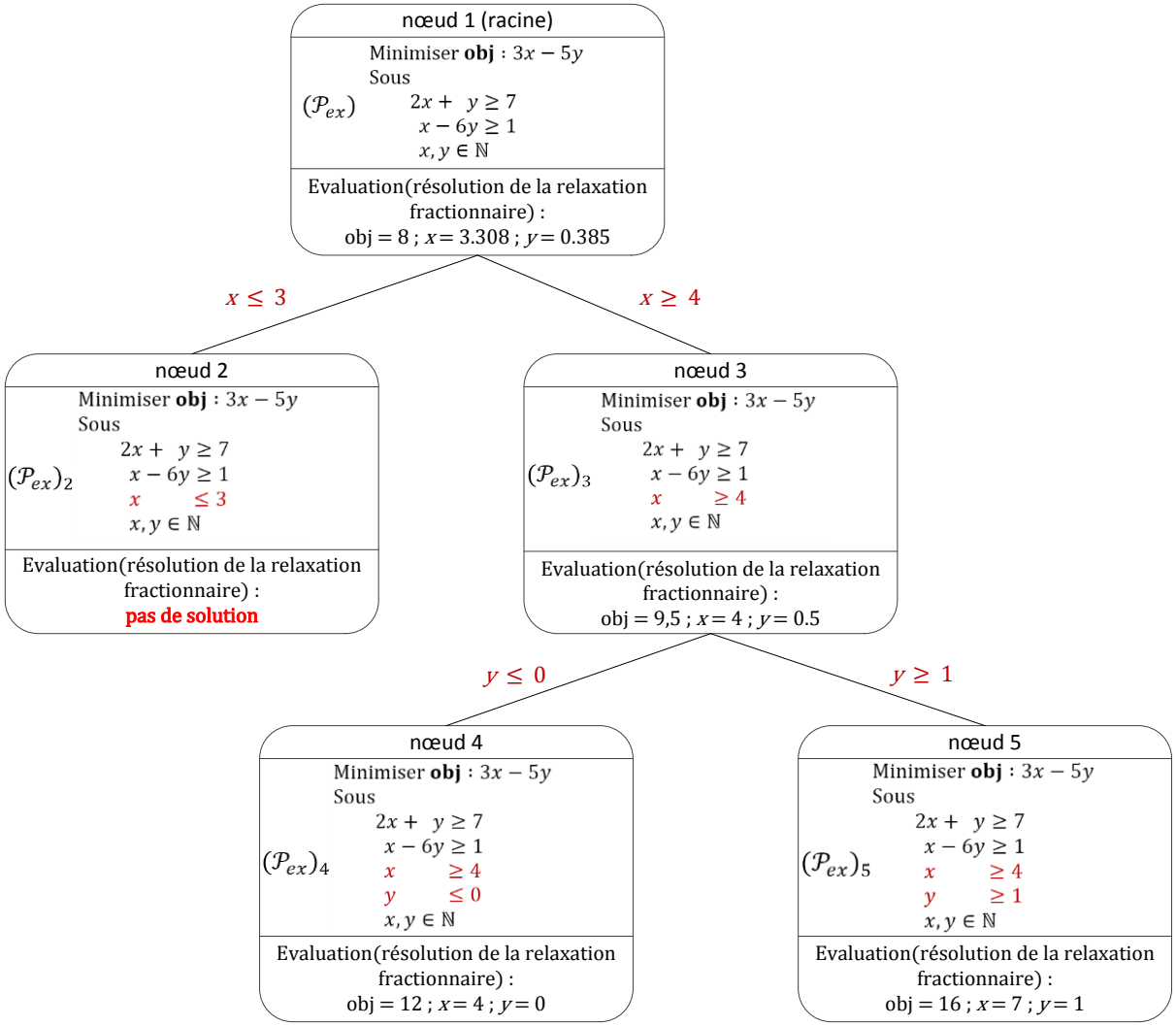


Figure 2: exemple d'un arbre de recherche obtenu en résolvant $(\mathcal{P}_{ex})$ par branch & bound

3.3 Le *branch* & *price*

Le *branch* & *price* est une méthode de résolution pour les PLNE contenant un très grand nombre de variables. Il consiste à associer un *branch* & *bound* et une génération de colonnes. L'idée est de résoudre les relaxations fractionnaires (phase évaluation du *branch* & *bound*) par génération de colonnes.

Contrairement au *branch* & *bound* qui peut être géré complétement par le solveur sans intervention de l'utilisateur, le *branch* & *price* nécessite le développement d'algorithmes *ad hoc* par l'utilisateur.

On peut mettre en évidence 3 phases principales du *branch* & *price* durant lesquelles l'utilisateur devra intervenir :

- **phase 1** : résolution du nœud courant (associé à la relaxation fractionnaire du problème maître courant) par génération de colonnes

- **phase 2** : séparation (recherche de variables pour le branchement et création de nouveaux nœuds)
- **phase 3** : propagation des contraintes de branchement dans le nouveau nœud courant

La Figure 3 présente un organigramme schématisant le déroulement d'un *branch* & *price*.

3.3.1 La phase de séparation

Un des points délicats dans la construction d'un *branch* & *price* est l'étape de séparation. On a vu (section 3.2.3) que la manière classique de réaliser la séparation dans un *branch* & *bound* est de choisir une variable fractionnaire dans le PL courant et d'ajouter une contrainte linéaire (appelée *contrainte de branchement*) liée à la valeur de cette variable dans chaque nouveau nœud fils.

Ces contraintes de branchement fournissent des variables duales qui doivent être prises en compte dans la phase de *pricing* (comme on l'a vu dans la section 3.1.2). Malheureusement il est généralement très difficile d'intégrer ces nouvelles variables duales au sous-problème de *pricing* (ceci est dû au fait que les contraintes de branchement ne représentent pas des contraintes du problème mais représentent simplement des décisions prises plus ou moins arbitrairement).

L'idée est alors d'adopter une règle de branchement qui ne modifie pas le problème maître (c'est-à-dire qui n'ajoute pas directement une contrainte linéaire dans le problème maître). Une règle de branchement bien connue est la règle de Ryan & Foster (voir [5]). Elle impose des restrictions sur les variables qui peuvent être générées durant le *pricing* et élimine certaines variables du problème maître. Aucune contrainte linéaire n'est ajoutée avec cette règle. Elle sera illustrée sur le problème de coloration de graphe dans la section 5.

3.3.2 *Branch* & *price* : algorithme général

Un *branch & price* se déroule exactement de la même façon qu'un *branch & bound* à ceci près que la résolution de la relaxation fractionnaire du PLNE courant se fait par génération de colonnes et que les contraintes de branchement doivent être spécifiques au problème traité.

La Figure 3 présente un organigramme schématisant le déroulement d'un *branch* & *price*.

Le déroulement détaillé d'un *branch* & *price* sera illustré sur le problème de coloration de graphe dans la section 5. La prochaine section présente le problème de coloration de graphe (*graph coloring*) et sa formulation linéaire. Il est important de bien comprendre en quoi consiste ce problème avant de passer à la section 5.

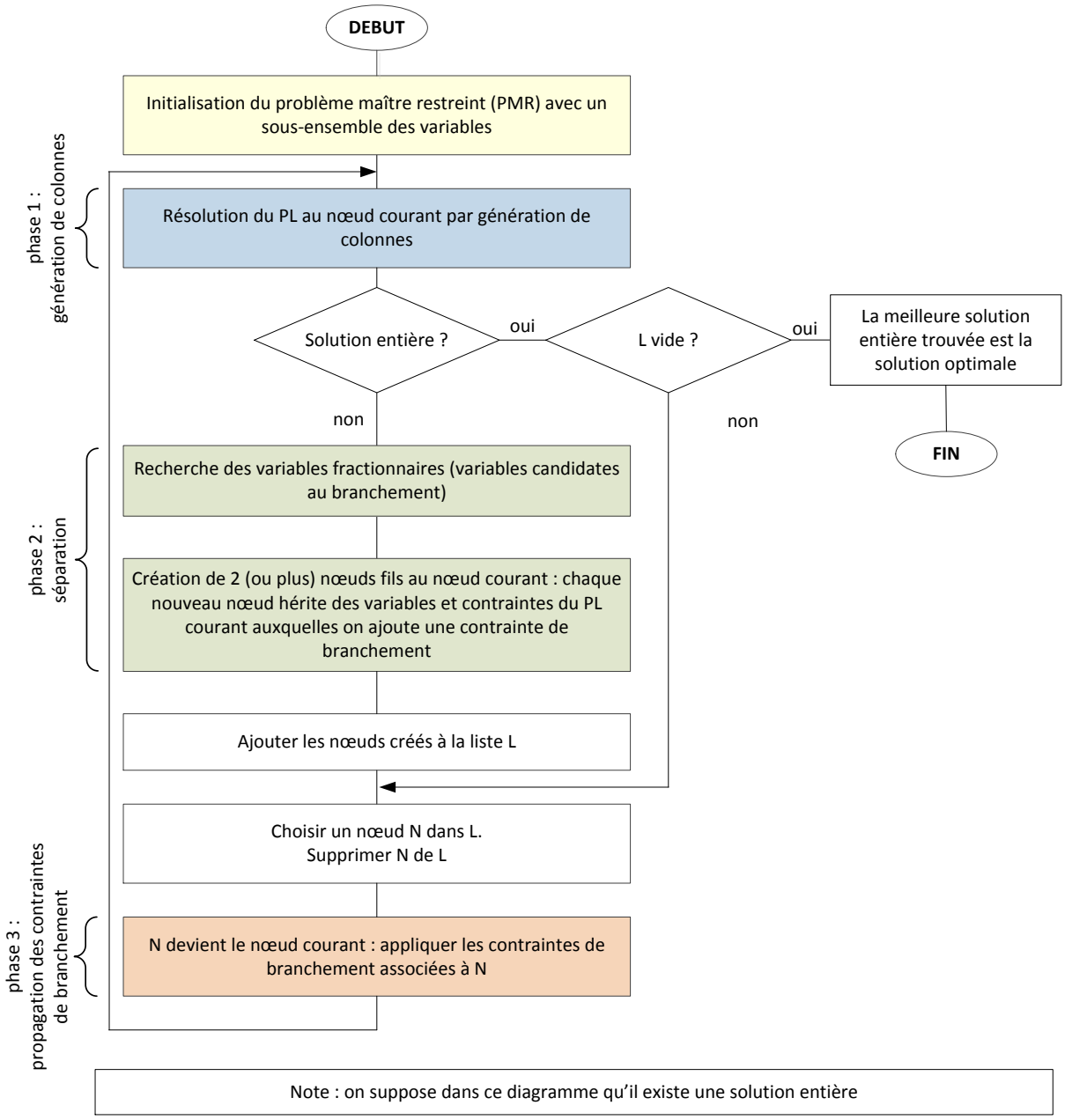


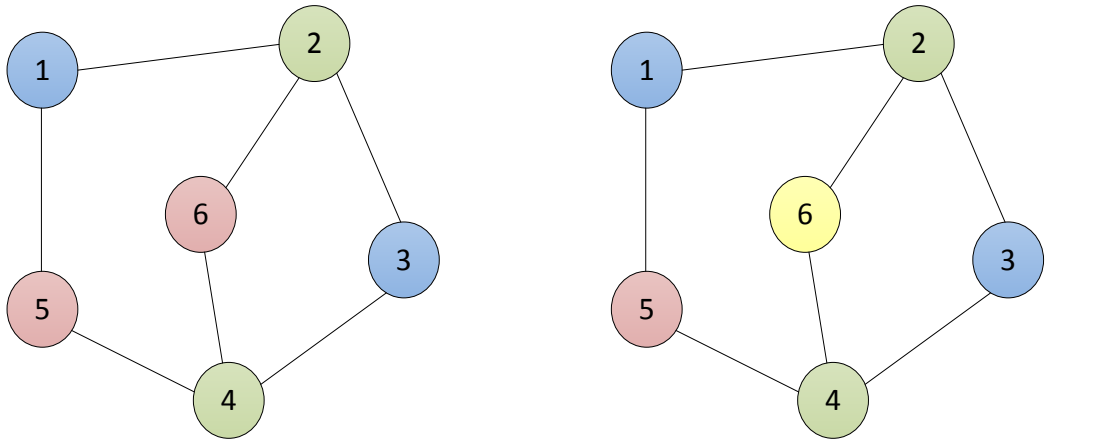
Figure 3. Schéma général d'un branch & price

4 Présentation du problème de coloration de graphe

4.1 Définition du problème

Soit $G = (X, E)$ un graphe. X est l'ensemble des sommets et E est l'ensemble des arcs.
Le problème de coloration consiste à trouver le nombre minimal de couleurs pour colorier les sommets du graphe G, de sorte que, pour chaque arc $(i, j) \in E$ le sommet i et le sommet j aient une couleur différente. Autrement dit, deux sommets voisins ne doivent pas avoir la même couleur.

La Figure 4 montre deux colorations possibles pour un même graphe. La première est clairement meilleure car elle n'utilise que trois couleurs (contre quatre pour la seconde).



(a) Exemple de coloration avec 3 couleurs (b) Exemple de coloration avec 4 couleurs

Figure 4. Deux coloriages différents pour un même graphe

Pour plus de détails sur ce problème voir l'article de Mehrotra et Trick ([3]).

4.2 Modèle linéaire adapté à la génération de colonnes

Il existe plusieurs modèles linéaires pour ce problème. On s'intéresse ici uniquement à celui qui est adapté à la génération de colonnes.

Commençons par constater que, dans une solution, un ensemble de sommets d'une même couleur constitue un stable (c'est-à-dire un ensemble de sommets 2 à 2 non voisins). Le problème de coloration consiste donc à trouver une partition du graphe en stables (chaque sommet doit ainsi apparaître dans un et un seul stable). Le nombre de couleurs est alors donné par le nombre de stables utilisés.

Par exemple, la Figure 4 (a) fait apparaître 3 stables : (1,3), (2,4) et (5,6) tandis que la Figure 4 (b) en fait apparaître 4 : (1,3), (2,4), (5) et (6).

Notons $\mathcal{S}$ l'ensemble de tous les stables du graphe G. On définit alors le programme linéaire de la manière suivante :

Variables de décision

$\forall s \in \mathcal{S}, x_s = \begin{cases} 1 \text{ si le stable s appartient à la solution} \\ 0 \text{ sinon} \end{cases}$

Contraintes

Un sommet i quelconque du graphe G doit être dans exactement un stable : $\sum_{s \in \mathcal{S} | i \in s} x_s = 1$.

Fonction objectif

On minimise le nombre de stables utilisés : $\sum_{s \in \mathcal{S}} x_s$

Modèle linéaire

$(\mathcal{P}_G)$	Minimiser $\sum_{s\in\mathcal{S}} x_s$
	Sous :
problème linéaire initial	$\forall i \in X, \sum_{s\in\mathcal{S}\mid i\in s} x_s = 1$
	$\forall s \in \mathcal{S}, x_s \in \{0,1\}$

Remarque : si on reprend les notations utilisées dans la section 2 pour définir le PL général $(\mathcal{P}_{init})$, on a :

- $n = |\mathcal{S}|$ (nombre de stables)
- $m = |X|$ (nombre de sommets du graphe G)
- $\forall i = [\![1, n]\!]$, $c_i = 1$ (chaque stable a un coefficient 1 dans la fonction objectif)
- $\forall j = [\![1, m]\!], \forall i = [\![1, n]\!], a_{ji} = 1$si le sommet j appartient au stable i, 0 sinon
- $\forall j = [\![1, m]\!], d_j = 1$

De plus les contraintes sont des égalités (et non des inégalités comme dans la définition générale). Cela ne change pas le calcul du coût réduit ni la méthodologie générale (les variables duales seront simplement définies dans tout $\mathbb{R}$ et non uniquement dans $\mathbb{R}^+$).

5 Résolution du problème de coloration de graphe par *branch* & *price*

L'objet de cette section est de montrer comment résoudre le modèle linéaire $(\mathcal{P}_G)$ de la section précédente par *branch* & *price*.

Le nombre de variables de $(\mathcal{P}_G)$ est égal au nombre de stables de G et il est très coûteux (voire impossible dans un temps "raisonnable") d'énumérer tous les stables d'un graphe (mise à part pour les graphes de très petite taille). On a donc besoin d'une méthode de résolution qui ne nécessite pas de connaître toutes les variables : le *branch* & *price* est donc tout à fait indiqué.

On reprend le graphe à 6 sommets et 7 arcs (voir Figure 5) de la section précédente pour illustrer les différentes étapes du *branch* & *price*.

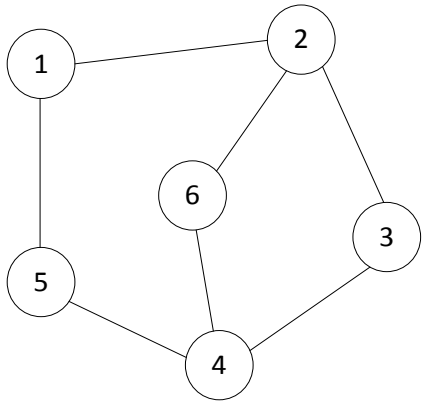


Figure 5 : graphe à 6 sommets utilisé pour illustrer les étapes du branch & price

5.1 Notations

Dans toute cette section on utilisera les notations suivantes (elles seront rappelées dans les sous-sections) :

Notations liées au graphe

- $G = (X, E)$: le graphe
- $X = \{1, \dots, |X|\}$: l'ensemble des sommets de G ($|X| = 6$ dans l'exemple de la Figure 5)
- E : l'ensemble des arcs de G

Notations liées au problème maître

- $(\mathcal{P}_G)$: problème initial théorique (avec toutes les variables)
- $(\mathcal{P}_G)_n$: problème maître restreint au nœud $n, (n \geq 1)$ avant la génération de colonnes (le nœud 1 correspond à la racine)
- $(\mathcal{P}_G)_{n.j}$: problème maître restreint au nœud $n, (n \geq 1)$ après génération de j nouvelles variables au cours l'évaluation du nœud n par génération de colonnes
- $(\mathcal{P}_G)_n^{\mathbb{R}}$: relaxation fractionnaire de $(\mathcal{P}_G)_n$
- λ_i : variable du duale associée à la $i^{ème}$ contrainte du problème maître, $i \in [\![1, |X|]\!]$ (au nœud courant)

Notations liées au sous-problème

- $(SP)_n$: sous problème associé à $(\mathcal{P}_G)_n$ (au nœud $n \geq 1$)
- $(z_i)_{i \in X}$: vecteur inconnu indexé sur les sommets du graphe (ce vecteur représente un stable : $z_i = 1$ signifie que le sommet i est dans le stable) dans le sous-problème courant
- $(\lambda_i)_{i \in X}$: vecteur des valeurs des variables duales du problème maître courant

5.2 Recherche des variables initiales

Dans ce problème de coloration de graphe, une variable est associée à un stable. Une solution très simple pour construire des variables initiales est donc d'utiliser les stables triviaux : ce sont les stables qui contiennent uniquement 1 sommet. On génère ainsi autant de variables initiales que de sommets.

Pour le graphe à 6 sommets de la Figure 4 on obtient donc : l'ensemble des stables initiaux $\mathcal{S}$ = {(1), (2), (3), (4), (5), (6)}, associés respectivement aux variables x_1, x_2, x_3, x_4, x_5 et x_6. A l'aide de ces 6 variables initiales on construit le problème maître restreint suivant :

$(\mathcal{P}_G)_1$

$$\text{Minimiser } x_1 + x_2 + x_3 + x_4 + x_5 + x_6$$

Sous :

$$x_1 = 1$$
$$x_2 = 1$$
$$x_3 = 1$$
$$x_4 = 1$$
$$x_5 = 1$$
$$x_6 = 1$$

$$\forall i \in \mathcal{S}, x_i \in \{0,1\}$$

On nomme $(\mathcal{P}_G)_n$ le PLNE associé au nœud $n, (n \geq 1)$. $(\mathcal{P}_G)_1$ est donc le PLNE associé à la racine de l'arbre. On notera $(\mathcal{P}_G)_n^{\mathbb{R}}$ la relaxation fractionnaire de $(\mathcal{P}_G)_n$.

5.3 Phase 1 : Génération de colonnes pour le nœud courant

Comme expliqué dans la section 3.3 le PLNE du nœud courant est évalué en résolvant sa relaxation fractionnaire par génération de colonnes.

5.3.1 Ecriture du sous-problème

On a vu dans la section 3.1.2 qu'une variable est améliorante si elle a un coût réduit strictement négatif. On souhaite donc générer un stable associé à une variable de coût réduit strictement négatif.
Cela peut être fait de manière heuristique ou de manière exacte en utilisant, par exemple, un programme linéaire décrit de la manière suivante :

Variables de décision :

$$\forall i \in X, z_i = \begin{cases} 1 \text{ si le sommet i est utilisé dans le stable} \\ 0 \text{ sinon} \end{cases}$$

Contraintes :
Les contraintes sont celles liées à la définition d'un stable : 2 sommets voisins ne peuvent pas faire partie du même stable :

$$\forall (i,j) \in E, z_i + z_j \leq 1$$

Fonction objectif :
On cherche un stable avec un coût réduit strictement négatif. Le coût réduit $\bar{c}_S$ d'un stable S quelconque se calcule à l'aide des variables $(z_i)_{i \in X}$ de la manière suivant :

$$\bar{c}_S = c_S - \sum_{i \in X} \lambda_i z_i$$

où :

- $c_S = 1$ (coefficient du stable S dans la fonction objectif du problème maître)
- $\forall i \in X, \lambda_i$ est la variable duale associée à la contrainte i du problème maître (et donc associée au sommet i)
- $(z_i)_{i \in X}$ décrit le stable : $\forall i \in X, z_i$ vaut 1 si le sommet i fait partie du stable, 0 sinon

On cherche donc un stable S tel que $\bar{c}_S < 0$, autrement dit tel que $\sum_{i \in X} \lambda_i z_i > c_S$. Le coefficient c_S étant constant ($= 1$), on maximise $\sum_{i \in X} \lambda_i z_i$.

Finalement on obtient le sous-problème suivant :

sous-problème (*pricing*)	Maximiser $\sum_{i \in X} \lambda_i z_i$ Sous : $\forall (i,j) \in E, z_i + z_j \leq 1$ $\forall i \in X, z_i \in \{0,1\}$

La stable généré est améliorant si $\sum_{i \in X} \lambda_i z_i > 1$.

Remarque : on a écrit le sous-problème à résoudre à la racine (pas de contrainte de branchement). La section 5.5 présente la manière d'écrire le sous-problème pour prendre en compte les contraintes de branchement.

5.3.2 Exemple

Génération d'une première colonne

Pour la génération de colonnes on travaille uniquement avec la relaxation fractionnaire du PLNE du nœud courant. Pour le nœud racine on considère donc la relaxation fractionnaire de $(\mathcal{P}_G)_1$ que l'on note $(\mathcal{P}_G)_1^{\mathbb{R}}$:

$(\mathcal{P}_G)_1^{\mathbb{R}}$

Minimiser $x_1 + x_2 + x_3 + x_4 + x_5 + x_6$

Sous :

$$x_1 = 1$$
$$x_2 = 1$$
$$x_3 = 1$$
$$x_4 = 1$$
$$x_5 = 1$$
$$x_6 = 1$$

$$\forall i \in [\![1,6]\!], x_i \in \mathbb{R}^+$$

La résolution de $(\mathcal{P}_G)_1^{\mathbb{R}}$ donne les variables duales suivantes : $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = \lambda_5 = \lambda_6 = 1$. On construit donc le sous-problème suivant :

$(SP)_1$

sous-problème associé à $(\mathcal{P}_G)_1^{\mathbb{R}}$

Maximiser $\sum_{i=1}^{6} z_i$

Sous :

$$z_1 + z_2 \leq 1$$
$$z_2 + z_3 \leq 1$$
$$z_3 + z_4 \leq 1$$
$$z_4 + z_5 \leq 1$$
$$z_1 + z_5 \leq 1$$
$$z_6 + z_2 \leq 1$$
$$z_6 + z_4 \leq 1$$

$$\forall i \in [\![1,6]\!], z_i \in \{0,1\}$$

La résolution du sous-problème donne $z_3 = z_5 = z_6 = 1$ et $z_1 = z_2 = z_4 = 0$. Ce qui correspond au stable (3, 5, 6) de coût $1 + 1 + 1 = 3$. 3 est strictement plus grand que 1, le stable est donc améliorant. On associe une nouvelle variable (x_7) à ce stable et on l'ajoute au problème maître $(\mathcal{P}_G)_1^{\mathbb{R}}$:

$(\mathcal{P}_G)_{1.1}^{\mathbb{R}}$

Minimiser $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$

Sous :

$$x_1 = 1$$
$$x_2 = 1$$
$$x_3 + x_7 = 1$$

$$x_4 = 1$$
$$x_5 + x_7 = 1$$
$$x_6 + x_7 = 1$$

$$\forall i \in [\![1,7]\!], x_i \in \mathbb{R}^+$$

Génération d'une seconde colonne

On continue la génération de colonne en résolvant $(\mathcal{P}_G)_1^{\mathbb{R}}$ dans lequel la nouvelle variable x_7 a préalablement été ajoutée. On obtient les variables duales : $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = \lambda_5 = 1$ et $\lambda_6 = -1$.

Dans le sous problème on change seulement la fonction objectif (λ n'intervient pas dans les contraintes) qui devient : Maximiser $z_1 + z_2 + z_3 + z_4 + z_5 - z_6$.

La résolution du sous-problème donne $z_2 = 1$ et $z_5 = 1$. On génère donc le stable (2, 5) (de coût 1+1=2 > 1) et on ajoute la variable x_8 associée dans le PL maître :

$(\mathcal{P}_{GC})_{1.2}^{\mathbb{R}}$

Minimiser $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$
Sous :
$$x_1 = 1$$
$$x_2 + x_5 = 1$$
$$x_3 + x_7 = 1$$
$$x_4 = 1$$
$$x_5 + x_7 + x_8 = 1$$
$$x_6 + x_7 = 1$$

$$\forall i \in [\![1,8]\!], x_i \in \mathbb{R}^+$$

Génération des colonnes suivantes

On continue la génération de colonnes jusqu'à ce qu'il n'y ait plus de variable améliorante. On génère ainsi encore 5 stables (2,4), (1,3), (1,6), (1,4) et (1,3,6) associés respectivement aux variables x_9 à x_{13}. L'ajout de ces variables au problème maître donne :

$(\mathcal{P}_G)_{1.7}^{\mathbb{R}}$

Minimiser $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$
Sous :
$$x_1 + x_{10} + x_{11} + x_{12} + x_{13} = 1$$
$$x_2 + x_5 + x_9 = 1$$
$$x_3 + x_7 + x_{10} + x_{13} = 1$$
$$x_4 + x_9 + x_{12} = 1$$
$$x_5 + x_7 + x_8 = 1$$
$$x_6 + x_7 + x_{11} + x_{13} = 1$$

$$\forall i \in [\![1,13]\!], x_i \in \mathbb{R}^+$$

Enfin la résolution de $(\mathcal{P}_G)_{1.7}^{\mathbb{R}}$ donne les variables duales : $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = \lambda_5 = 0.5$ et $\lambda_6 = 0$.

Il n'existe pas de variable améliorante. La génération de colonnes pour le nœud racine s'arrête.

La solution de $(\mathcal{P}_G)^{\mathbb{R}}_{1.7}$ est $x_7 = x_8 = x_9 = x_{12} = x_{13} = 0.5$ (les autres variables sont à 0). C'est donc une solution fractionnaire de coût 2.5. On entre donc dans la phase 2 (séparation) pour poursuivre la résolution du problème initial$(\mathcal{P}_G)$.

Le Tableau 1 présente une synthèse des variables et les stables associés.

variables initiales	stable associé	variables générées	stable associé
x_1	(1)	x_7	(3,5,6)
x_2	(2)	x_8	(2,5)
x_3	(3)	x_9	(2,4)
x_4	(4)	x_{10}	(1,3)
x_5	(5)	x_{11}	(1,6)
x_6	(6)	x_{12}	(1,4)
		x_{13}	(1,3,6)

Tableau 1. Récapitulatif des variables au nœud racine

5.3.3 Génération d'une variable dans le cas où le PL est infaisable

Durant le *branch* & *price* il est possible que les contraintes de branchement contraignent tellement le problème qu'il n'y a alors plus de solution (le PMR devient irréalisable). Dans ce cas on peut poursuivre la résolution en utilisant les multiplicateurs de Farkas (voir section 3.1.2) pour essayer de générer une nouvelle variable qui permettrait de lever l'irréalisabilité du problème.

Notons $(\mu_i)_{i\in X}$ les multiplicateurs de Farkas. On remplace dans le sous-problème les valeurs duales par les multiplicateurs de Farkas (les contraintes restent les mêmes). Le sous-problème de génération d'un nouveau stable s'écrit alors :

sous-problème (*pricing*)	Maximiser $\sum_{i\in X} \mu_i z_i$ Sous : $\forall (i,j) \in E, z_i + z_j \leq 1$ $\forall i \in X, z_i \in \{0,1\}$

Le stable généré est utile si $\sum_{i\in X} \mu_i z_i > 0$. Dans le cas contraire il est inutile de l'ajouter au PMR et on peut en déduire que le PLNE associé au nœud courant n'a pas de solution.

5.4 Phase 2 : Séparation (règle de branchement de Ryan & Foster)

5.4.1 Principe

Comme expliqué dans la section 3.3.1, il faut éviter d'ajouter des contraintes de branchement linéaires au problème maître. On utilise donc ici la règle de branchement de Ryan & Foster qui s'énonce ainsi pour la coloration de graphe (voir justification dans [3]) :
"si la solution est fractionnaire alors on peut en déduire qu'il existe deux stables S_1 et S_2 ainsi que deux sommets i_1 et i_2 tels que :

$i_1 \in S_1 \cap S_2$ ***et*** $i_2 \in S_1 - S_2$ ***et*** *au moins une des deux variables x_{S_1} ou x_{S_2} est fractionnaire."*

L'idée est alors de créer les 2 contraintes de branchement (C_1) et (C_2) suivantes :

- (C_1) : les sommets i_1 et i_2 sont dans le même stable (*ils ont la même couleur*)
- (C_2) : les sommets i_1 et i_2 sont dans des stables différents (*ils ont des couleurs différentes*)

Le nœud courant est donc séparé en deux nœuds fils, reprenant chacun l'intégralité des variables et contraintes du nœud courant auxquelles on ajoute la contrainte (C_1) pour le premier fils et (C_2) pour le second.

Dans les nœuds fils, les variables associées à des stables ne respectant pas les nouvelles contraintes de branchement sont mises à 0 (elles ne sont plus considérées pendant la résolution).

5.4.2 Exemple

On reprend l'exemple de la section 5.3.2 et on continue la résolution.

La solution de $(\mathcal{P}_G)_{1.7}^{\mathbb{R}}$ est $x_7 = x_8 = x_9 = x_{12} = x_{13} = 0.5$. Autrement dit les stables suivant sont utilisés à 50% dans la solution : (3,5,6), (2,5) (2,4), (1,4), (1,3,6).

Pour la séparation on peut par exemple utiliser les stables $S1 = (3,5,6)$ et $S2 = (2,5)$ et le couple de sommets (3,5). En effet, les sommets 3 et 5 sont tous deux associés à des variables fractionnaires, $S1$ contient les sommets 3 et 5 tandis que $S2$ contient 5 mais pas 3. On en déduit les contraintes de branchement suivantes :

- (C_1) : les sommets 3 et 5 sont dans le même stable (*ils ont la même couleur*)
- (C_2) : les sommets 3 et 5 sont dans des stables différents (*ils ont des couleurs différentes*)

et on crée les deux nœuds fils suivants :

1er fils (nœud 2)

$(\mathcal{P}_G)_2$

Minimiser $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$

Sous :

$$x_1 + x_{10} + x_{11} + x_{12} + x_{13} = 1$$
$$x_2 + x_5 + x_9 = 1$$
$$x_3 + x_7 + x_{10} + x_{13} = 1$$
$$x_4 + x_9 + x_{12} = 1$$
$$x_5 + x_7 + x_8 = 1$$
$$x_6 + x_7 + x_{11} + x_{13} = 1$$

"les sommets 3 et 5 sont dans le même stable"

$\forall i \in [\![1,13]\!], x_i \in \{0,1\}$

2sd fils (nœud 3)

$(\mathcal{P}_G)_3$

Minimiser $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$

Sous :

$$x_1 + x_{10} + x_{11} + x_{12} + x_{13} = 1$$
$$x_2 + x_5 + x_9 = 1$$
$$x_3 + x_7 + x_{10} + x_{13} = 1$$
$$x_4 + x_9 + x_{12} = 1$$
$$x_5 + x_7 + x_8 = 1$$

$$x_6 + x_7 + x_{11} + x_{13} = 1$$

"les sommets 3 et 5 sont dans des stables différents"

$$\forall i \in [\![1,13]\!], x_i \in \{0,1\}$$

La section suivante montre comment prendre en compte ces contraintes de branchement notamment lors du pricing.

5.5 Phase 3 : propagation des contraintes de branchement

5.5.1 Principe

Lorsque l'on souhaite évaluer un nouveau nœud, il faut au préalable prendre en compte les contraintes de branchement associées à ce nœud. La prise en compte des contraintes de branchement se fait en 2 étapes :

1ère étape : les variables associées à des stables qui ne respectent pas les contraintes de branchement sont fixées à 0 (cela revient à les supprimer du problème maitre restreint).

2ème étape : Le sous problème est transformé pour prendre en compte les contraintes de branchement : en effet, il faut éviter de générer un nouveau stable qui ne respecte pas les contraintes de branchement.

Pour cela, notons $DIFF$ l'ensemble des couples de sommets qui ne doivent pas être dans un même stable et $EGAL$ l'ensemble des couples de sommets qui doivent être dans un même stable. Le sous-problème devient alors :

sous-problème général (*pricing*) prenant en compte les contraintes de branchement

$$\text{Maximiser} \sum_{i \in X} \lambda_i z_i$$

Sous :

$$\forall (i,j) \in E, \quad z_i + z_j \leq 1$$
$$\forall (i,j) \in DIFF, \quad z_i + z_j \leq 1$$
$$\forall (i,j) \in EGAL, \quad z_i - z_j = 0$$
$$\forall i \in X, z_i \in \{0,1\}$$

5.5.2 Exemple 1

On reprend la résolution de l'exemple précédent (section 5.4.2).

5.5.2.1 Prise en compte des contraintes de branchement dans le nœud 2

Imaginons que l'on veuille évaluer le nœud 2 associé à la contrainte de branchement "les sommets 3 et 5 sont dans le même stable".

étape 1 : les variables associées à des stables contenant le sommet 3 sans le sommet 5 ou le sommet 5 sans le sommet 3 sont fixées à 0 : c'est le cas de x_3, x_5, x_8, x_{10}, x_{13}. On fixe donc $x_3 = x_5 = x_8 = x_{10} = x_{13} = 0$.

Après avoir fixé ces variables à 0, $(\mathcal{P}_G)_2$ est résolu. On obtient la solution $x_1 = x_7 = x_9 = 1$ de coût 3. La solution duale est $\lambda_1 = \lambda_2 = \lambda_5 = 1$. On doit maintenant générer de nouvelles colonnes, c'est l'objet de l'étape 2.

étape 2 : on construit le sous-problème $(SP)_2$. On doit interdire la génération d'un stable qui contient le sommet 3 sans le sommet 5 (ou le sommet 5 sans le sommet 3). On ajoute donc dans $(SP)_2$ la contrainte $z_3 - z_5 = 0$.

$(SP)_2$ Maximiser $\sum_{i \in X} \lambda_i z_i$

Sous :

$\forall (i,j) \in E, z_i + z_j \leq 1$ [contraintes liées à la définition d'un stable]

$z_3 - z_5 = 0$ [contrainte de branchement : 3 et 5 ensemble]

$\forall i \in X, z_i \in \{0,1\}$

La solution de $(SP)_2$ est $z_1 = 1$ de coût 1. Le coût n'étant pas strictement plus grand que 1 on en déduit que la colonne associée à la solution n'est pas améliorante. La génération de colonnes s'arrête. La solution du PMR étant entière il n'y a pas de phase de séparation : on est sur une feuille de l'arbre.

5.5.2.2 Prise en compte des contraintes de branchement dans le nœud 3

On poursuit l'exemple et on évalue le nœud 3 associé à la contrainte de branchement "les sommets 3 et 5 ne sont pas dans le même stable".

étape 1 : on fixe donc x_7 à 0. On résout $(\mathcal{P}_G)_3$, on obtient la solution $x_4 = x_8 = x_{13} = 1$ de coût 3. La solution duale est $\lambda_3 = \lambda_4 = \lambda_5 = 1$.

étape 2 : on résout le sous-problème $(SP)_3$ dans lequel on a ajouté la contrainte $z_3 + z_5 \leq 1$ qui interdit d'avoir à la fois les sommets 3 et 5 dans la solution.
La solution de $(SP)_3$ est $z_3 = 1$ de coût 1. Cette solution ne représente donc pas une colonne améliorante. La génération de colonnes s'arrête. De même que pour le nœud 2 on est sur une feuille de l'arbre.

$(SP)_3$ Maximiser $\sum_{i \in X} \lambda_i z_i$

Sous :

$\forall (i,j) \in E, z_i + z_j \leq 1$ [contraintes liées à la définition d'un stable]

$z_3 + z_5 \leq 1$ [contrainte de branchement : 3 et 5 séparés]

$\forall i \in X, z_i \in \{0,1\}$

Remarque 1 : on voit bien à l'aide de cet exemple que les contraintes de branchement sont déplacées dans le sous problème. Ceci évite de les avoir dans le problème maître sous forme linéaire, ce qui induirait des variables duales que l'on ne sait pas prendre en compte dans le sous-problème.

Remarque 2 : une seule décision de branchement a été prise pour arriver au nœud 2 (rspt 3), c'est pourquoi dans $(SP)_2$ (rspt $(SP)_3$) on n'a qu'une seule contrainte de branchement. Dans le cas général, on a plusieurs contraintes de branchement : les contraintes de branchement se cumulent de père en fils (voir exemple suivant section 5.5.3).

5.5.2.3 Résultat final

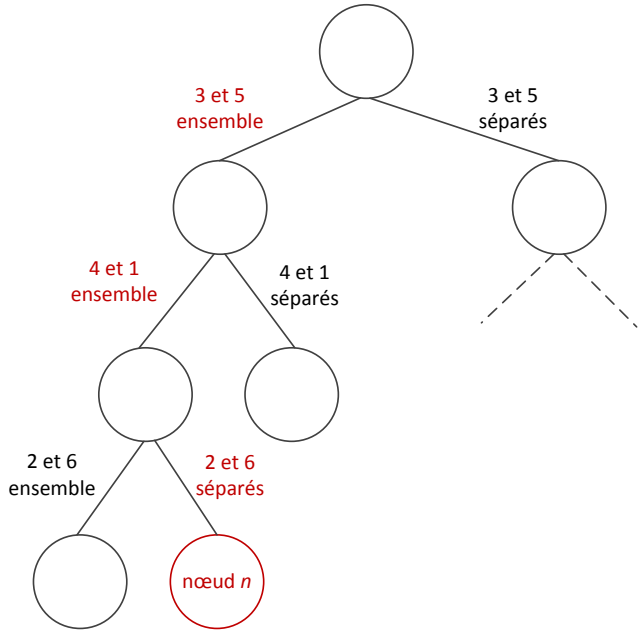
Les nœuds 2 et 3 donnent des solutions entières. On a donc construit un arbre de recherche avec seulement 3 nœuds (l'exemple étant volontairement très simple). Les deux solutions entières trouvées au cours de l'exploration arborescente ont le même coût ($= 3$). Elles sont donc toutes les 2 optimales.

5.5.3 Exemple 2

Supposons disposer d'un graphe avec plus de sommets et qui demande donc la construction d'un arbre plus grand. Imaginons être arrivé à un nœud n dans l'arbre de recherche auquel sont associées les contraintes de branchement suivantes :

- les sommets 3 et 5 sont ensemble
- les sommets 4 et 1 sont ensemble
- les sommets 2 et 6 ne sont pas ensemble

Le sous-problème de ce nœud n doit alors prendre en compte toutes ces contraintes. La Figure 6 donne un exemple d'arbre et le sous-problème lié au nœud n.



(a)

Maximiser $\sum_{i \in X} \lambda_i z_i$

Sous :

$$\forall (i,j) \in E, z_i + z_j \leq 1$$
$$z_3 - z_5 = 0$$
$$z_4 - z_1 = 0$$
$$z_2 + z_6 \leq 0$$
$$\forall i \in X, z_i \in \{0,1\}$$

(b)

Figure 6 (a) : un arbre de recherche ; (b) : le sous-problème associé au nœud n

Les rappels théoriques concernant le *branch & price* s'achèvent ici. Les prochaines sections concernent l'implémentation d'un *branch* & *price* à l'aide de la librairie SCIP.

Partie 2

Programmer un *branch* & *price* avec SCIP : description des classes de SCIP et exemple de programme complet

6 Programmation d'un *branch* & *price* avec SCIP (en C++)

SCIP est un *framework* qui permet de faire du *Branch Cut & Price* (BCP). Il est écrit en C et possède une interface en C++ (qu'il n'est pas obligatoire d'utiliser). Il est conçu de telle sorte que l'utilisateur puisse modifier facilement son comportement pendant la résolution d'un PLNE pour ajouter, par exemple, des coupes, des variables, des heuristiques, modifier les règles de branchement, etc...

La modification du comportement par défaut passe par l'ajout de plugins. Un plugin modifie une étape de la résolution en donnant la main à l'utilisateur au moment où cette étape se déroule (par exemple, au moment du branchement pour que l'utilisateur puisse choisir lui-même les variables sur lesquelles brancher).

En C++ Il existe une classe par plugin. Il suffit alors à l'utilisateur d'en dériver une classe fille et de redéfinir les méthodes de la classe mère qu'il souhaite utiliser. La méthode redéfinie sera appelée automatiquement par SCIP lors de la résolution au moment opportun.

Chaque phase du *branch* & *price* décrite dans la section 3.3 est pilotée par une classe de SCIP :

- phase 1 (génération de colonnes) : classe ObjPricer (http://scip.zib.de/doc/html/classscip_1_1ObjPricer.php)

- phase 2 (séparation) : classe ObjBranchrule (http://scip.zib.de/doc/html/classscip_1_1ObjBranchrule.php)

- phase 3 (prise en compte des contraintes de branchement) : classe ObjConshdlr http://scip.zib.de/doc/html/classscip_1_1ObjConshdlr.php

Dans chaque classe un certain nombre de méthodes sont virtuelles pures car elles doivent absolument être redéfinies par l'utilisateur s'il décide de créer une classe fille.

Pour utiliser ces classes il faut donc :
1. dériver une classe fille et redéfinir (au minimum) les fonctions virtuelles pures,
2. informer SCIP qu'on l'on souhaite inclure le plugin correspondant (pour chaque classe il existe une fonction **`SCIPinclude<nomDeLaClasseMere>(...)`** que nous verrons plus loin).

Remarque : il faut bien évidemment redéfinir les méthodes en gardant exactement les mêmes paramètres d'entrée que ceux de la méthode mère. Pour des raisons de lisibilité je ne liste pas systématiquement les paramètres d'entrée des méthodes. Ils sont donnés dans les exemples de code ou dans la documentation de référence de SCIP. D'une manière générale il n'est pas nécessaire d'avoir

connaissance de ces paramètres pour comprendre le fonctionnement de la méthode, de plus ils ne sont - pour la plupart - généralement pas utilisés dans le corps de la méthode.

Les sections suivantes détaillent l'utilisation des trois classes qui servent à programmer un *branch* & *price*. Le point délicat est de comprendre à quel moment sont appelées les différentes méthodes au cours de la résolution, pour cela on pourra se référer au schéma général donné à la fin de cette partie (Figure 12).

6.1 Phase 1 : génération de colonnes (classe `ObjPricer`)

La classe **`ObjPricer`** est la classe à dériver pour faire de la génération de colonnes avec SCIP (que ce soit dans le cadre d'un *branch* & *price* ou d'une génération de colonnes sur un problème fractionnaire).

Trois méthodes sont à redéfinir dans cette classe :

- **`scip_init`** : cette fonction est appelé après le prétraitement réalisé automatiquement par SCIP sur le problème initial (et avant la résolution proprement dite). En effet SCIP peut réaliser des modifications sur le problème initial pour des raisons d'efficacité (suppression des variables redondantes par exemple). Il est alors nécessaire de travailler dans cette classe *pricer* avec les variables et contraintes modifiées. Cette fonction permet d'accéder aux pointeurs sur les variables et contraintes modifiées.

- **`scip_redcost`** : cette fonction est appelée lors de la recherche d'une variable améliorante. Dans cette fonction l'utilisateur doit :
  1. récupérer les variables duales
  2. résoudre le sous problème
  3. injecter la /les nouvelle(s) variable(s) dans le problème maître (s'il y en a)

- **`scip_farkas`** : cette fonction est utilisée dans le cas d'un problème maitre restreint non faisable. La démarche est la même que celle utilisée dans **`scip_redcost`**, à ceci près qu'il faut utiliser les multiplicateurs de Farkas (et non les variables duales qui n'existent pas dans un PL infaisable).

- L'Exemple de code 1 montre la définition minimale d'une classe fille dérivée de la classe fille **`ObjPricer`**. Le constructeur de la classe prend en paramètre d'entrée un pointeur sur l'environnement SCIP (qui doit donc avoir été défini par l'utilisateur au préalable) et le nom du *pricer* (il est possible de créer plusieurs *pricers* avec des noms différents). Voir l'exemple de code complet ([6]) pour l'implémentation des méthodes dans le cas de la coloration de graphe.

```
class MonPricer : public ObjPricer
{
public:
        Pricer(SCIP * scip, const char * p_name);

        virtual ~Pricer();

        // récupération des pointeurs sur les variables et contraintes transformées (après le
           prétraitement automatique réalisé par SCIP)
        virtual SCIP_RETCODE scip_init(SCIP* scip, SCIP_PRICER* pricer);

        // recherche d'une variable améliorante et insertion dans le problème maitre si trouvée
```

```
        virtual  SCIP_RETCODE scip_redcost(SCIP* scip, SCIP_PRICER* pricer, SCIP_Real* lowerbound,
                                           SCIP_Bool* stopearly, SCIP_RESULT* result);

        // recherche d'une variable pour la réalisabilité et insertion dans le problème maitre si
           trouvée
        virtual  SCIP_RETCODE scip_farkas(SCIP* scip, SCIP_PRICER* pricer, SCIP_RESULT* result);
};
```

Exemple de code 1. Définition minimale d'une classe dérivée de **`ObjPricer`**

Pour que le plugin soit effectivement pris en compte par SCIP lors de la résolution il faut le créer et l'inclure dans SCIP avant l'appel à la fonction `SCIPsolve()` qui lance la résolution complète du PL ou PLNE. L'Exemple de code 2 montre comment faire.

```
// création d'un pointeur sur l'environnement SCIP
SCIP * scip;

[...] // création du modèle etc...

// création d'un pricer
MonPricer * pricer = new MonPricer(scip, "PricerColor");

// inclusion du plugin
SCIPincludeObjPricer(scip, pricer, true);

// activation du plugin (le pricer est le seul plugin qui nécessite d'être activé)
SCIPactivatePricer(scip, SCIPfindPricer(scip, "PricerColor"));


// résolution du modèle : les fonctions redéfinies par l'utilisateur seront automatiquement appelées
SCIPsolve(scip);
```

Exemple de code 2. Création et inclusion du plugin pour la génération de colonnes

L'appel à **`SCIPsolve()`** déclenche toute la mécanique de résolution de SCIP. Si un plugin a été ajouté alors il est intégré à la résolution. L'organigramme Figure 7 montre l'enchaînement des principales opérations effectuées par SCIP lors d'un appel à **`SCIPsolve()`**, sur un PL fractionnaire, quand l'utilisateur a inclus un *pricer*. Les parties en gris correspondent à des traitements ou des tests qui sont réalisés de manière automatique par SCIP. Les rectangles bleus correspondent à des traitements qui doivent être fait (par l'utilisateur) à l'intérieur d'une méthode de la classe fille de **`ObjPricer`**.

L'organigramme permet de comprendre à quel moment les différentes méthodes sont appelées au cours de la résolution. Dans le cas d'un PLNE ce mécanisme de génération de colonnes est effectué à chaque nœud, sur la relaxation fractionnaire du PLNE associé au nœud.

Remarque : Dans les organigrammes on fait apparaître le nom de la classe mère pour que la classe de SCIP à utiliser apparaisse clairement. Bien évidemment c'est la méthode redéfinie dans la classe dérivée qui est effectivement appelée.

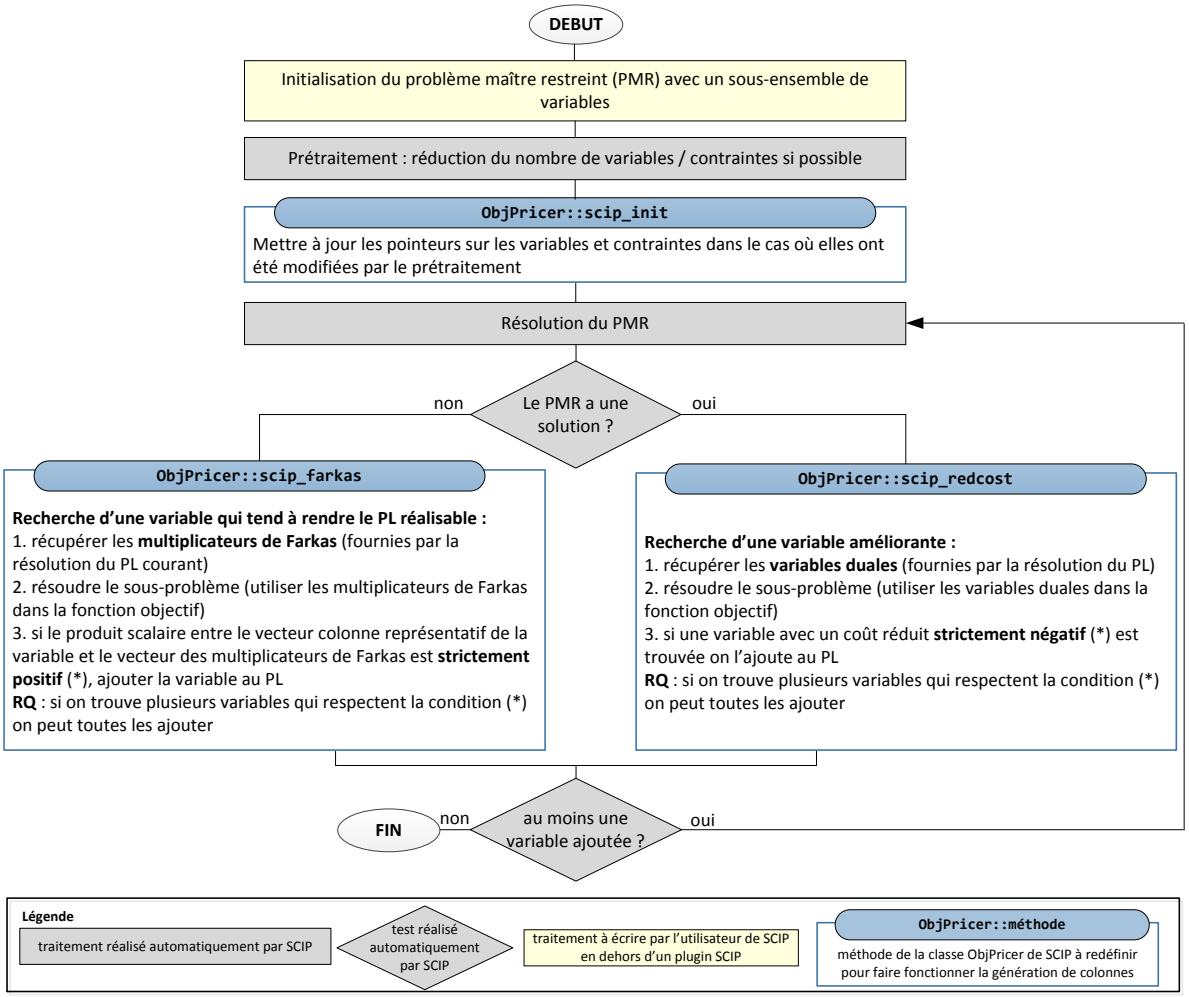


Figure 7. Principales opérations effectuées par SCIPsolve(), sur un PL fractionnaire, quand l'utilisateur a inclus un pricer

6.2 Phase 2 : Séparation (classe `ObjBranchrule`)

La classe **`ObjBranchrule`** est la classe à dériver pour définir sa propre règle de branchement. Elle comporte une seule méthode à redéfinir : **`scip_execlp`**.

Le corps de cette méthode doit comporter les éléments suivants :

- recherche des variables fractionnaires dans la solution courante,
- choix de la (ou des) variable(s) sur lesquelles brancher,
- création des nœuds fils et des contraintes de branchement associées.

L'Exemple de code 3 montre la définition minimale d'une classe dérivée de **`ObjBranchrule`** (pour l'implémentation voir l'exemple de code complet [6]). L'Exemple de code 4 montre comment inclure ce plugin pour qu'il soit pris en compte par SCIP lors de la résolution.

```
class BranchRule : public ObjBranchrule
{
public:
```

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

        ~BranchRule() {}

        virtual SCIP_RETCODE scip_execlp(
                SCIP*              scip,
                SCIP_BRANCHRULE*   branchrule,
                SCIP_Bool          allowaddcons,
                SCIP_RESULT*       result);
};
```

Exemple de code 3. Définition minimale d'une classe dérivée de ObjBranchrule

```
// création d'un pointeur sur l'environnement SCIP
SCIP * scip;

[...] // création du modèle etc...

// définition d'un pointeur
BranchRule* branch_ptr = new BranchRule(scip, "MyBranchrule", 500000, -1, 1);

// inclusion du plugin branchrule dans SCIP
SCIPincludeObjBranchrule(scip, branch_ptr, true);
```

Exemple de code 4. Création et inclusion du plugin de branchement

Le fonctionnement général de la classe **`ObjBranchrule`**, dans le cadre d'un *branch* & *bound* classique (pas de génération de colonnes), est décrit par l'organigramme de la Figure 8.

Remarque : La mise à jour des bornes inférieures et supérieures est gérée automatiquement par SCIP, elle n'est pas explicitée dans cet organigramme afin de ne pas le surcharger.

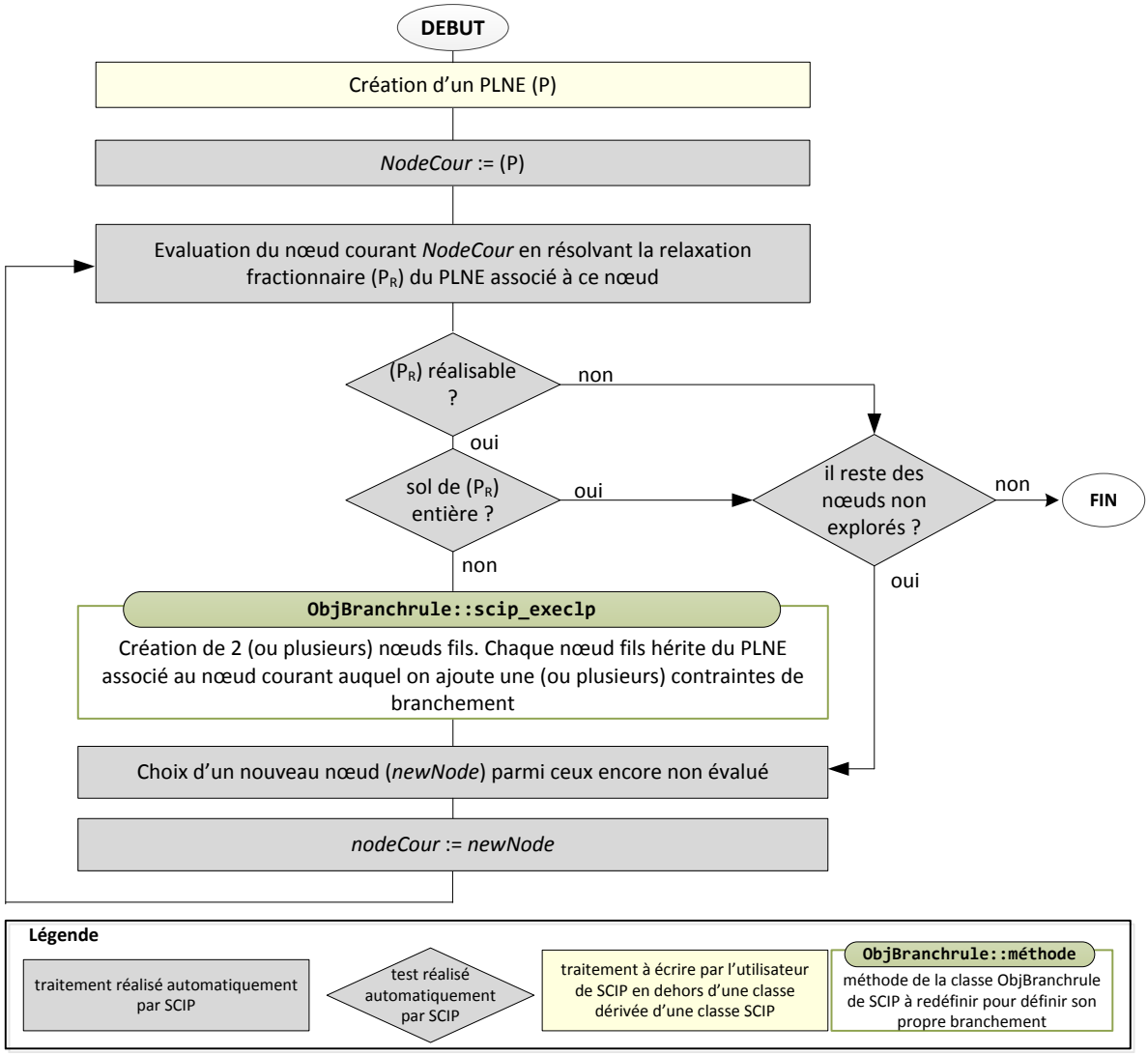


Figure 8. Fonctionnement du Branch & Bound de SCIP quand la règle de branchement est définie par l'utilisateur (dérivation de la classe ObjBranchRule)

6.3 Phase 3 : propagation des contraintes de branchement (classe `ObjConsHdlr`)

Dans la section précédente on a vu que, pour définir une règle de branchement personnalisée, l'utilisateur doit créer lui-même les nœuds fils et les contraintes de branchement associées.

Comment sont gérées ces contraintes de branchement dans SCIP ?

6.3.1 1er cas : les contraintes de branchement sont des contraintes linéaires

Si les contraintes de branchement sont des contraintes linéaires alors l'utilisateur peut les ajouter directement dans le modèle (en précisant qu'il s'agit de contraintes locales afin de les différencier des contraintes initiales du modèle). Elles seront activées automatiquement et prises en compte lors de l'évaluation du nœud (l'utilisateur n'a rien d'autre à faire).

6.3.2 2sd cas : les contraintes de branchement ne sont pas linéaires

La gestion de contraintes de branchement linéaires est très simple pour l'utilisateur, malheureusement on a vu dans la section 3.3.1 que, dans le cadre d'un *branch* & *price*, ce type de contraintes n'est pas adapté.

SCIP dispose d'un plugin (le *Constraint handler,* la classe associée étant **`ObjConsHdlr`**) qui permet à l'utilisateur de gérer toutes sortes de contraintes (linéaires ou non linéaires). C'est ce plugin qu'il faut utiliser pour gérer les contraintes de branchement lors d'une résolution par *branch* & *price*.

Le rôle du *constraint handler* est de gérer des contraintes (que ce soit dans un PL ou PLNE) qui ne peuvent pas être intégrées directement dans le modèle initial :
- soit parce qu'elles sont non linéaires,
- soit parce qu'elles sont trop nombreuses pour être énumérées.

Ces deux cas ne se traitent pas de la même manière (ce ne sont pas les mêmes méthodes de la classe qui interviennent). On s'intéresse ici uniquement au premier cas, l'implémentation de la règle de branchement de Ryan & Foster nécessitant la manipulation de contraintes non linéaires.

Dans le cas de la coloration de graphe le *constraint handler* que l'on va définir implémente donc les contraintes "2 sommets ont la même couleur" et "2 sommets n'ont pas la même couleur".

La classe **`ObjConsHdlr`** possède 31 méthodes mais (heureusement) il n'est pas obligatoire de toutes les redéfinir. En réalité seulement 3 méthodes nous intéressent pour implémenter la règle de Ryan & Foster :

- **`scip_active`** : cette fonction est appelée quand SCIP entre dans le nœud fils du nœud courant (on dit que le nœud fils est *activé*)

- **`scip_deactive`** : cette fonction est appelée quand SCIP sort d'un nœud fils et retourne dans le nœud père (on dit que le nœud fils est *désactivé*)

- **`scip_prop`** : cette fonction est appelée juste après l'activation d'un nœud si de nouvelles contraintes associées à ce nœud doivent être prises en compte

Avant d'expliquer comment utiliser ces méthodes pour prendre en compte les contraintes de branchement il faut bien comprendre comment SCIP gère le parcours de l'arbre. SCIP effectue, par défaut, un parcours " *best first search* " : un mécanisme interne à SCIP donne des priorités aux nœuds au moment où ils sont créés, les nœuds sont alors évalués dans l'ordre défini par les priorités.

SCIP est donc amené à effectuer des "sauts" dans l'arbre en cours de résolution (en effet le prochain nœud à évaluer peut se trouver dans une branche différente de celle du nœud courant). Quand SCIP passe d'un nœud à un autre il suit le plus court chemin dans l'arbre entre ces deux nœuds, suivant ce chemin il désactive et active les nœuds en conséquence. Les fonctions **`scip_active`** et **`scip_deactive`** permettent alors à l'utilisateur d'agir à chaque activation / désactivation d'un nœud.

SCIP numérote les nœuds dans l'ordre dans lequel ils sont créés (en commençant à 1 pour la racine). Par exemple, si on considère l'arbre de la Figure 9, on en déduit que les nœuds ont été évalués de la manière suivante :

- le nœud 1 est évalué (phase d'évaluation), les nœuds 2 et 3 sont créés (phase de séparation)
- puis le nœud 2 est évalué et les nœuds 4 et 5 sont créés
- puis le 4 est évalué et les nœuds 6 et 7 sont créés
- puis le 3 est évalué et les nœuds 8 et 9 sont créés
- puis le 9 est évalué et les nœuds 10 et 11 sont créés
- puis le 10 est évalué et les nœuds 12 et 13 sont créés

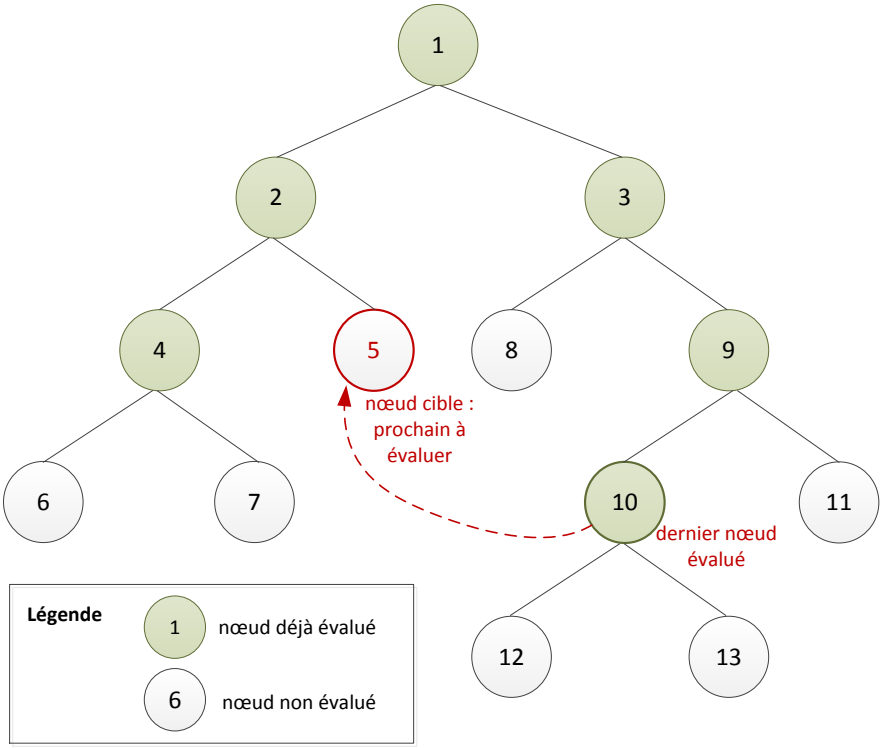


Figure 9: exemple d'arbre de recherche créé par SCIP

Imaginons que SCIP décide, après l'évaluation du nœud 10, d'aller évaluer le 5. Les activations et désactivations de nœuds se font alors dans l'ordre suivant :

1. désactivation du nœud 10
2. désactivation du nœud 9
3. désactivation du nœud 3
4. activation du nœud 2
5. activation du nœud 5

A chaque activation d'un nœud la fonction **`scip_active`** est appelée avec les paramètres correspondant au nœud activé (notamment la liste des contraintes locales à ce nœud). De même, à chaque désactivation d'un nœud la fonction **`scip_deactive`** est appelée. L'utilisateur, qui aura pris soin de redéfinir ces deux méthodes, peut alors faire les modifications nécessaires pour prendre en compte de manière personnalisée les contraintes gérées par le *Constraint Handler*.

Le déroulement d'un *branch* & *bound* (classique, sans génération de colonnes) avec un *constraint handler* est donné par l'organigramme de la Figure 10. On suppose dans cet organigramme que seules les 3 méthodes évoquées précédemment ont été redéfinies.

Remarque : Il existe bien sûr un plugin permettant à l'utilisateur de choisir lui-même la priorité des nœuds et le type de parcours, mais ce n'est pas l'objet de ce document et cela ne changerait pas le mécanisme général de résolution.

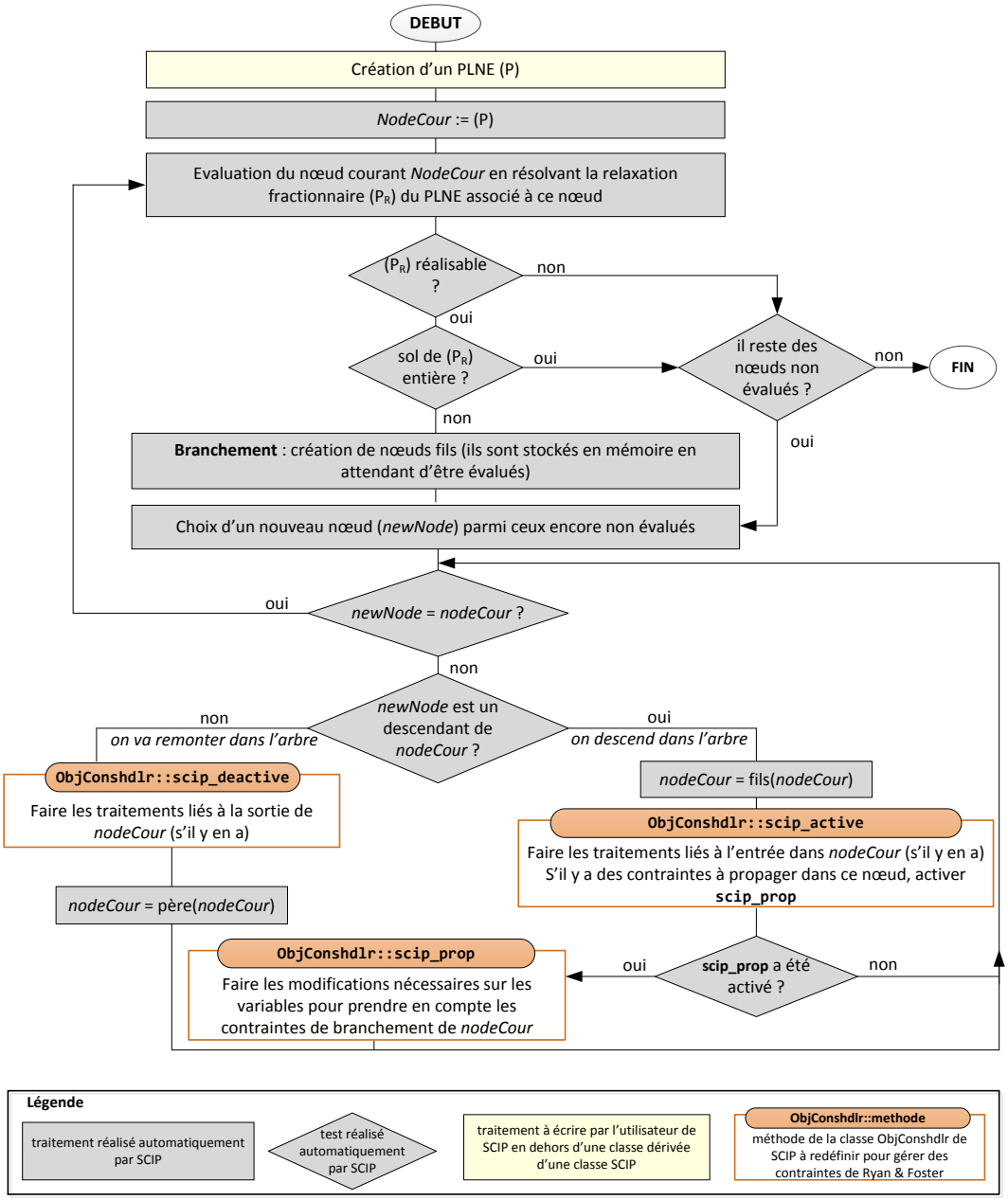


Figure 10. Résolution d'un PLNE par SCIP quand on inclut le plugin du Constraint Handler en redéfinissant seulement les trois méthodes utiles à la gestion des contraintes Ryan & Foster

L'Exemple de code 5 montre la définition minimale d'une classe dérivée de **`ObjConsHdlr`**. La classe **`ObjConsHdlr`** comporte des méthodes virtuelles pures qu'il est donc obligatoire de redéfinir mais qui n'ont pas d'utilité dans le cas où on l'utilise pour implémenter la règle de Ryan & Foster. Ces méthodes

sont donc redéfinies mais ne font rien. La liste de leurs paramètres d'entrée n'est pas explicitée afin de ne pas trop charger l'exemple (voir l'exemple complet [6]).

```
class BranchConsHdlr : public ObjConshdlr
{
public:

       //constructeur
       BranchConsHdlr(SCIP * scip) : ObjConshdlr(scip, "BranchConsHdlr", "stores the
local branching decisions", 0, 0, 9999999, -1, 1, 1, 0, FALSE, FALSE, TRUE,
SCIP_PROPTIMING_BEFORELP, SCIP_PRESOLTIMING_FAST | SCIP_PRESOLTIMING_EXHAUSTIVE){}

       // destructeur
       virtual ~BranchConsHdlr(){}

       // propage la ctr (c'est ici que la ctr est rendue "effective")
       virtual SCIP_RETCODE scip_prop(SCIP* scip, SCIP_CONSHDLR* conshdlr, SCIP_CONS**
            conss, int nconss, int nusefulconss, int nmarkedconss, SCIP_PROPTIMING
            proptiming, SCIP_RESULT* result);

       // méthode appelée quand on active un noeud
       virtual SCIP_RETCODE scip_active(SCIP * scip, SCIP_CONSHDLR * conshdlr,
            SCIP_CONS * cons);

       // méthode appelée quand on désactive un noeud
       virtual SCIP_RETCODE scip_deactive(SCIP * scip, SCIP_CONSHDLR * conshdlr,
            SCIP_CONS * cons);

       // cette méthode doit être appelée pour libérer la mémoire du consdata (si on en
           a un !)
       virtual SCIP_RETCODE scip_delete(SCIP * scip, SCIP_CONSHDLR * conshdlr,
             SCIP_CONS * cons, SCIP_CONSDATA ** consdata);

       // les 7 fonctions suivantes doivent redéfinies car virtuelles pures dans la
         classe mère mais elles ne servent pour implémenter Ryan & Foster
       virtual SCIP_RETCODE scip_trans(...){}


       virtual SCIP_RETCODE scip_check(SCIP* scip, SCIP_CONSHDLR* conshdlr, SCIP_CONS**
            conss, int nconss, SCIP_SOL* sol, SCIP_Bool checkintegrality, SCIP_Bool
            checklprows, SCIP_Bool printreason, SCIP_Bool completely, SCIP_RESULT*
            result){
              return SCIP_OKAY; }

       virtual SCIP_RETCODE scip_enfolp(...) {
              return SCIP_OKAY; }

       virtual SCIP_RETCODE scip_enfops(...){
              return SCIP_OKAY;}

       virtual SCIP_RETCODE scip_lock(...){
              return SCIP_OKAY;}

       virtual SCIP_RETCODE scip_sepalp(...) {
              return SCIP_OKAY;}

       virtual SCIP_RETCODE scip_sepasol(...) {
              return SCIP_OKAY;}
};
```

Exemple de code 5. Définition minimale d'une classe dérivée de ObjConshdlr pour faire fonctionner la règle de Ryan & Foster, les arguments des 7 dernières méthodes sont remplacées par ..., voir [6] pour la définition complète

```
// création d'un pointeur sur l'environnement SCIP
SCIP * scip;

[...] // création du modèle etc...

// création d'un pointeur sur la classe fille de ObjConsHdlr
BranchConsHdlr * consHdlr_ptr = new BranchConsHdlr(scip);

// inclusion du plugin dans SCIP
SCIPincludeObjConshdlr(scip, consHdlr_ptr, true);
```

Exemple de code 6. Inclure une classe dérivée de ObjConsHdlr

6.3.3 Utiliser le *constraint handler* pour implémenter la règle de Ryan & Foster

La section 6.3.2 présente les méthodes du *constraint handler* dont on a besoin pour implémenter la règle de branchement de Ryan & Foster. Comment les utilise-t-on concrètement ?

Concrètement pour implémenter cette règle de branchement il faut :

- dans **`scip_active`** : récupérer la contrainte de branchement associée au nœud en cours d'activation (elle est passée en paramètre de la fonction par SCIP). On la stocke dans une Structure De Données (SDD) *ad hoc* (construite par l'utilisateur : une pile ou un tableau par exemple) de sorte d'y avoir facilement quand on sera dans la méthode **`redcost`** du pricer (en effet durant le pricing on doit connaître l'ensemble des contraintes de branchement qui s'appliquent au nœud courant)

- dans **`scip_deactive`** : supprimer la dernière contrainte ajoutée dans la SDD lors de l'appel à **`scip_active`**

- dans **`scip_prop`** : récupérer la contrainte de branchement associée au nœud courant (elle est passée en paramètre de la fonction par SCIP) et fixer à 0 toutes les variables qui ne respectent pas cette règle. Il existe une fonction SCIP (**`SCIPfixVar`**) pour fixer à 0 une variable. Fixer une variable x à 0 n'est pas la même chose que d'ajouter une contrainte de la forme $x = 0$. En effet dans le premier cas SCIP résout la relaxation en faisant comme si les variables fixées à 0 n'existaient pas alors que dans le second cas on ajoute une contrainte (qui donnera donc une variable duale à considérer dans le *pricing*).

On remarque que l'implémentation de la règle de branchement de Ryan & Foster nécessite de stocker dans une structure de donnée (SDD) dédiée (et créée par l'utilisateur) toutes les décisions de branchement qui ont été prises pour arriver à un nœud donnée.
Cela ne nécessite qu'une seule SDD : au fur et à mesure qu'on descend dans l'arbre on ajoute les décisions de branchement dans cette SDD (les décisions de branchement se cumulent de père en fils). Quand on remonte dans l'arbre la dernière décision de branchement est supprimée de la SDD. Cette structure fonctionne donc comme une pile : "dernier arrivé, premier sorti". A partir de maintenant on parlera donc de pile pour désigner la SDD qu'on utilise (même si on peut utiliser autre chose).

La Figure 11 (a) montre un arbre de recherche possible lorsqu'on résout un problème de *graph coloring*. La Figure 11 (b) présente la pile associée au nœud 10 de cet arbre après qu'il ait été activé : la pile contient toutes les décisions de branchement qui ont été prises pour arriver au nœud 10 depuis la racine.

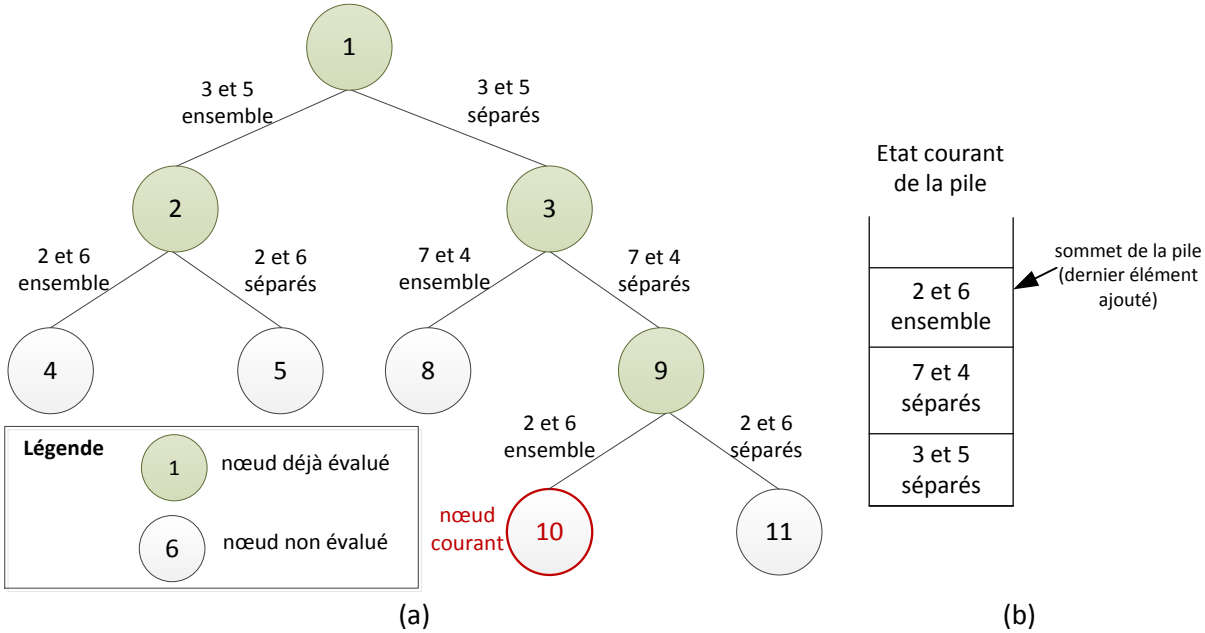


Figure 11 : (a) un arbre de recherche A ; (b) état de la pile après activation du nœud 10 de l'arbre A

Remarque : un même nœud peut être activé plusieurs fois à des moments différents de l'exploration arborescente. En effet, SCIP effectuant des "sauts" dans l'arbre on peut être amené à activer à nouveau un nœud déjà évalué car on redescend dans une branche qui avait été partiellement explorée. Par exemple sur l'arbre précédant (Figure 11 (a)), SCIP peut décider d'évaluer le nœud 10 puis le 5 puis le 11. Quand il passe du nœud 5 au nœud 11 il désactive alors les nœuds 5 et 2 puis active les nœuds 3 et 9 (qui avaient déjà été activés quand le nœud 10 était évalué).

6.4 Synthèse : interaction entre les différents modules

Le schéma de la Figure 12 montre comment s'articulent les 3 plugins précédents lorsqu'ils sont utilisés ensemble pour résoudre un PLNE par *branch* & *price* avec la règle de branchement de Ryan & Foster.

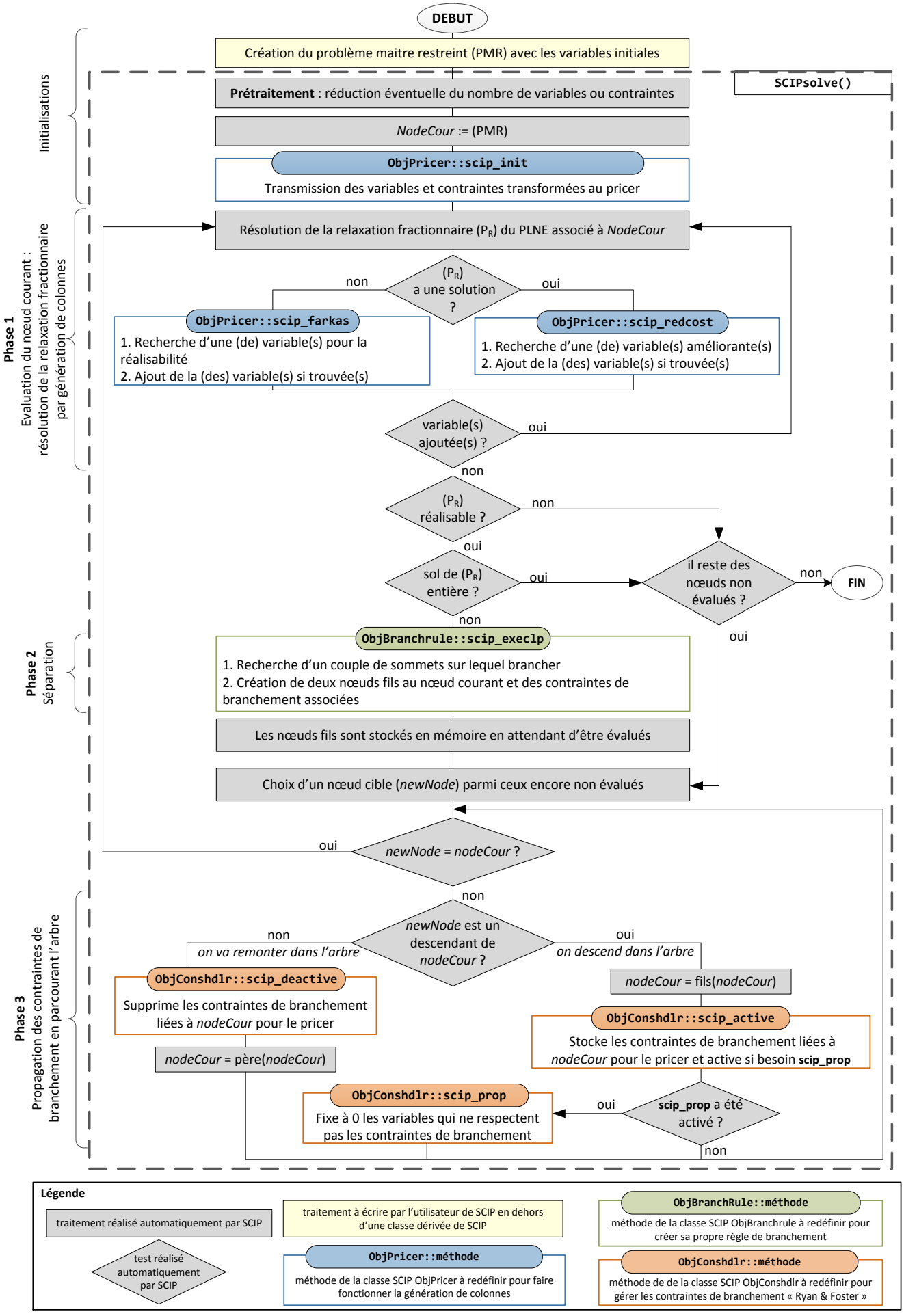


Figure 12. Schéma général du déroulement d'un branch & price incluant une règle de branchement "Ryan &Foster"

7 Programme complet en C++ / SCIP

Le programme complet ([6]), codé en C++ et utilisant la librairie SCIP 4.0, est disponible. Il est fourni avec un makefile pour être recompilé sous linux. Il a été testé avec gcc 4.8.5.

7.1 Organisation général du code source

- un fichier main (**`main.cpp`**)
- deux fichiers (1 .h + 1 .cpp) par classe SCIP à dériver :
    - **`Pricer.h`** et **`Pricer.cpp`** pour la classe `Pricer` dérivée de `ObjPricer`
    - **`BranchRule.h`** et **`BranchRule.cpp`** pour la classe dérivée de `ObjBranchrule`
    - **`BranchConsHdlr.h`** et **`BranchConsHdlr.cpp`** pour la classe dérivée de `ObjConshdlr`
    - deux fichiers **`Instance.h`** et **`Instance.cpp`** pour la classe utilisateur `Instance` qui lit et stocke l'instance
    - deux fichiers **`Master.h`** et **`Master.cpp`** pour la classe utilisateur `Master` qui stocke la définition du problème maitre (modèle linéaire) et assure les modifications d'ajout d'une nouvelle colonne
- deux fichiers d'entête supplémentaires : **`Stable.h`** permet de stocker un stable et **`SCIP_ConsData.h`** permet de stocker des données liées aux contrainte de branchement.

La section suivante détaille davantage le rôle des différentes classes.

7.2 Les différentes classes

7.2.1 La classe Instance

La classe `Instance` contient le graphe représenté par liste d'adjacence.

Son rôle est de lire un fichier d'instance et de stocker le graphe en mémoire.

7.2.2 La classe Master

La classe `Master` stocke les données relatives au problème maître :

- les variables (le nombre de variables augmente au fur et à mesure de la résolution) et les stables associés,
- les contraintes du problème (le nombre de contraintes est fixe mais de nouvelles variables sont ajoutées dans les contraintes au fur et à mesure de la résolution),
- les contraintes de branchement : la liste des contraintes de branchement est mise à jour à chaque mouvement dans l'arbre (descente vers un nœud fils ou remontée vers le nœud père). De cette manière, l'ensemble des contraintes de branchement stockées dans cette classe correspond toujours aux contraintes qui s'appliquent au nœud courant.

Le rôle de cette classe est de :

- initialiser le problème maître avec un sous ensemble de variables
- lancer la résolution
- récupérer la meilleure solution

La Figure 13 donne une modélisation UML de la classe Master et montre ses interactions avec les autres classes et structures.

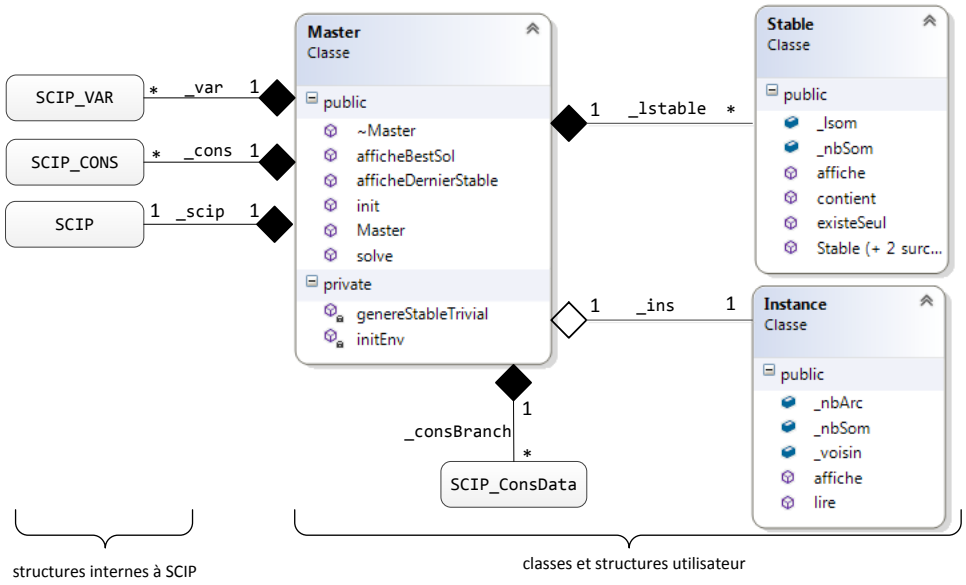


Figure 13 : environnement de la classe Master

7.2.3 La classe Stable

La classe `Stable` stocke les données d'un stable.

Elle contient les fonctions d'affichage d'un stable, de recherche d'un couple de sommets ou d'un sommet isolé dans un stable.

7.2.4 La classe Pricer

La classe `Pricer` stocke un pointeur sur le problème maître. En effet, pour mener à bien le pricing, on a besoin de connaître la valeur des variables duales de la relaxation du problème maître au nœud courant. Pour obtenir cette information on a besoin d'avoir un pointeur sur les contraintes (contraintes initiales et contraintes de branchement).

Le rôle de cette classe est de :

- créer et résoudre le sous-problème en utilisant les valeurs des variables duales (ou les multiplicateurs de Farkas si la relaxation du problème maitre restreint est infaisable),
- déterminer s'il existe une variable améliorante (ou qui tend à rendre le problème faisable),
- ajouter la variable trouvée.

La Figure 14 donne une modélisation UML de la classe `Pricer` et montre ses interactions avec les autres classes utilisateurs.

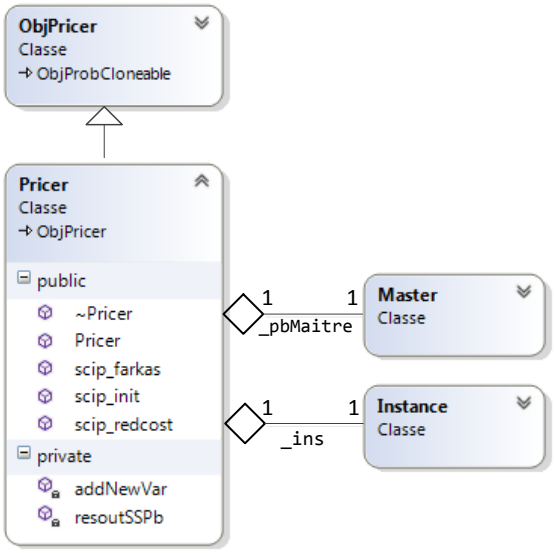


Figure 14 : environnement de la classe Pricer

7.2.5 La classe BranchRule

Les méthodes de la classe `BranchRule` sont exécutées au moment du branchement : c'est-à-dire quand la génération de colonnes se termine à un nœud donné et que la solution de la relaxation fractionnaire du problème maitre restreint (PMR) n'est pas entière.

Elle possède un pointeur sur le problème maître car elle a besoin d'accéder aux variables.

Son rôle est de

- rechercher les variables qui ont une valeur fractionnaire dans la solution courante (i.e. la solution de la relaxation fractionnaire du PMR du nœud courant) et de choisir un couple de sommets du graphe sur lequel brancher suivant la règle de Ryan & Foster (voir section 5.4),
- créer les nœuds fils et les contraintes de branchement associées.

7.2.6 La classe BranchConsHdlr

La classe `BranchConsHdlr` permet d'appliquer la contrainte de branchement aux nœuds. Son rôle est de permettre l'ajout / la suppression de contraintes de branchement dans la SDD ad hoc, à l'aide de fonctions SCIP d'activation et désactivation de nœuds. Le but étant que, après l'activation d'un nœud, la SDD contienne l'ensemble des contraintes de branchement qui s'appliquent au nœud courant.

7.2.7 La structure SCIP_ConsData

La structure `SCIP_ConsData` permet de stocker les informations des contraintes de branchement. C'est une structure SCIP que l'on peut attacher à une contrainte, les informations qu'elle contient seront alors accessible dès lors que l'on possède un pointeur sur la contrainte. C'est pratique dans le cadre des fonctions d'activation / désactivation d'un nœud de la classe `BranchConsHdlr` car les pointeurs sur les contraintes de branchement sont passés directement par SCIP en paramètre de la fonction.

On stocke donc dans cette structure toutes les informations nécessaires à l'activation / désactivation d'un nœud (les indices de variables sur lesquels on branche, le type de contrainte, ...)

8 Points d'amélioration du *branch* & *price*

Afin de ne pas surcharger les explications, chaque étape du branch & price a été présentée de la manière la plus simple possible sans chercher l'efficacité concernant le temps de calcul ou le nombre de variables générées. Cependant, en pratique, il existe un certains nombres d'améliorations à apporter à la méthode pour être efficace.

D'une manière générale, il est judicieux dans une résolution par *branch & price* de :

- Initialiser le problème maître avec des variables non triviales (en utilisant des solutions générées de manière heuristiques par exemple),

- Utiliser une ou plusieurs heuristiques pour résoudre le sous-problème. En effet il est très coûteux de résoudre de manière exact le sous-problème à chaque fois. En général on utilise une / plusieurs heuristique(s) pour chercher une variable améliorante et on appelle la méthode exacte uniquement si la / les heuristique(s) n'en trouve(nt) pas,

- Ne pas inclure dans le sous-problème les sommets liés à des variables duales strictement négatives. En effet de tels sommets ne peuvent pas faire partie d'un stable améliorant (ils ne participent pas à rendre le coût réduit strictement négatif).

Dans le cas particulier du problème de coloration de graphe on peut envisager de :

- Modéliser le problème en utilisant uniquement des stables maximums (au sens de l'inclusion). En effet le nombre de stables maximums est généralement nettement inférieur au nombre de stables. Il y a donc, a priori, moins de variables à générer. Par contre cela change légèrement le modèle et la solution : dans ce cas un sommet pourrait avoir plusieurs couleurs (car il peut apparaitre dans plusieurs stables), il suffit alors d'en choisir une.

- Ne pas stocker les contraintes de branchement dans une SDD annexe pour les inclure ensuite dans le sous-problème, mais faire directement les modifications sur le graphe : si deux sommets doivent avoir la même couleur on les fusionne, si deux sommets doivent avoir des couleurs différentes on ajoute un arc entre les deux. De cette manière c'est seulement le graphe à partir duquel on construit le sous-problème qui est modifié et le *pricer* n'a plus besoin d'intégrer les contraintes de branchement.

9 Références

[1] J. Teghem. **Programmation linéaire**. Editions de l'université de Bruxelles, Editions Ellipses. 2004.

[2] V. Chvátal. **Linear Programming**. Freeman, New York 1983.

[3] A. Mehrotra and M.A. Trick, **A Column Generation Approach For Graph Coloring**, INFORMS Journal on Computing, 1995, volume 8, pages 344-354

[4] H. Toussaint**, Introduction au Branch Cut and Price et au solveur SCIP (Solving Constraint Integer Programs)**, rapport de recherche LIMOS RR13-07, 2013, http://fc.isima.fr/~toussain/doc/rapportBCP.pdf

[5] D. M. Ryan and B. A. Foster: **An Integer Programming Approach to Scheduling, In Computer scheduling of public transport: Urban passenger vehicle and crew scheduling**, A. Wren editor, North-Holland 1981, 269-280.

[6] Hélène Toussaint, Exemple de programme utilisant SCIP pour résoudre le problème de coloration de graphe. Programme complet à télécharger (2017) : http://fc.isima.fr/~toussain/doc/SCIP/exempleBranchPrice.zip