**Université Pierre et Marie Curie**

**Master Androide**

# Mise en œuvre d'un algorithme de Branch-and-Cut avec le framework SCIP

Pierre Fouilhoux
pierre.fouilhoux@lip6.fr

26 mars 2015

# Table des matières

**1 Cas d'étude : l'Optimisation Combinatoire** **3**
1.0.1 Problèmes d'Optimisation Combinatoire . . . . . . . . . . . . . 3
1.0.2 Problèmes classiques en optimisation combinatoire . . . . . . . . 3

**2 Programmation Linéaire en Nombres Entiers** **6**
2.1 Définitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Difficulté de la PLNE . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Relaxation continue et solveurs . . . . . . . . . . . . . . . . . . . 8
2.4 Modélisation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.4.1 Un puissant outil de modélisation . . . . . . . . . . . . . . . 9
2.4.2 Modélisation par un PLNE . . . . . . . . . . . . . . . . . . . 9
2.4.3 Linéarisation . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.5 Formulations classiques . . . . . . . . . . . . . . . . . . . . . . . . 12
2.5.1 Le problème du sac-à-dos . . . . . . . . . . . . . . . . . . . . 12
2.5.2 Recouvrement, pavage et partition . . . . . . . . . . . . . . . 13
2.5.3 Le problème du stable . . . . . . . . . . . . . . . . . . . . . 13
2.5.4 Le problème du voyageur de commerce . . . . . . . . . . . . . 14
2.5.5 Le problème de coloration . . . . . . . . . . . . . . . . . . . 17
2.6 Rappels théoriques sur la résolution . . . . . . . . . . . . . . . . . 18
2.6.1 PLNE compact . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.6.2 Algorithme de coupes et branchements . . . . . . . . . . . . . 19

**3 Mise en œuvre : exemple du sous-graphe acyclique induit** **25**
3.1 Le problème du sous-graphe acyclique induit . . . . . . . . . . . . . 25
3.2 Programmes et instances . . . . . . . . . . . . . . . . . . . . . . . 27
3.3 Structures de données pour un graphe . . . . . . . . . . . . . . . . 28
3.4 PLNE compact . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.5 PLNE non compacte : Logiciel SCIP . . . . . . . . . . . . . . . . . 29
3.5.1 Installation de SCIP . . . . . . . . . . . . . . . . . . . . . . 29
3.5.2 Description d'un programme SCIP . . . . . . . . . . . . . . . 30

3.5.3 La classe Constraint Handler . . . . . . . . . . . . . . . . . . . . . 30
3.5.4 Le Constraint Handler AcyclicCte . . . . . . . . . . . . . . . . . . 31
3.5.5 Autres classes optionnelles . . . . . . . . . . . . . . . . . . . . . . 31
3.5.6 Les fonctions importantes . . . . . . . . . . . . . . . . . . . . . . 32

**4 Solutions approchées et garantie 34**
4.1 Trouver des solutions réalisables . . . . . . . . . . . . . . . . . . . . . . 34
4.2 Solution à garantie théorique . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 Solution approchée avec garantie expérimentale . . . . . . . . . . . . . . 36

# Chapitre 1

# Cas d'étude : l'Optimisation Combinatoire

### 1.0.1 Problèmes d'Optimisation Combinatoire

Un problème d'optimisation combinatoire (OC) consiste à déterminer un plus grand (petit) élément dans un ensemble fini valué. En d'autres termes, étant donnés une famille $\mathcal{F}$ de sous-ensembles d'un ensemble fini $E = \{e_1, ...e_n\}$ et un système de poids $w = (w(e_1), ..., w(e_n))$ associé aux éléments de $E$, un problème d'optimisation consiste à trouver un ensemble $F \in \mathcal{F}$ de poids $w(F) = \sum_{e \in F} w(e)$ maximum (ou minimum), *i.e.*

$$\max \text{ ou } \min \ \{w(F) \mid F \in \mathcal{F}\}.$$

La famille $\mathcal{F}$ représente donc les solutions du problème. Elle peut correspondre à un ensemble de très grande taille que l'on ne connaît que par des descriptions ou des propriétés théoriques qui ne permettent pas facilement son énumération.

### 1.0.2 Problèmes classiques en optimisation combinatoire

#### Le problème du sac-à-dos

Considérons $n$ objets, notés $i = 1, ..., n$, apportant chacun un bénéfice $c_i$ mais possédant un poids $a_i$. On veut ranger ces objets dans un "sac" que l'on veut au maximum de poids $b$. Le problème de *sac-à-dos* (*knapsack*) consiste à choisir les objets à prendre parmi les $n$ objets de manière à avoir un bénéfice maximal et respecter la contrainte du poids à ne pas dépasser. Chaque objet $i$, $i \in \{1, ..., n\}$, doit être sélectionné au moins $p_i$ fois et au plus $q_i$ fois.
Ce problème se rencontre bien entendu dès que l'on part en randonnée en voulant emmener le plus possible d'objets utiles (nourriture, boissons,...). Mais ce problème est

plus fréquemment utilisé pour remplir les camions de transport, les avions ou bateaux de fret et même pour gérer la mémoire d'un microprocesseur.

### Recouvrement, pavage et partition

Soit $E = \{1, ..., n\}$ un ensemble fini d'éléments. Soit $E_1, ..., E_m$ des sous-ensembles de $E$. A chaque ensemble $E_j$ on associe un poids $c_j$, $j = 1, ..., m$.
Une famille $F \subseteq \{E_1, ..., E_m\}$ est dite
- un *recouvrement* de $E$ si $\cup_{E_j \in F} E_j = E$, pour tout $j \in \{1, ..., m\}$,
- un *pavage* de $E$ si $E_j \cap E_k = \emptyset$, pour tout $j \neq k \in \{1, ..., m\}$,
- une *partition* de $E$ si $F$ est à la fois un recouvrement et un pavage.

Prenons par exemple 5 éléments $E = \{1, 2, 3, 4, 5\}$ et 4 sous-ensembles $E_1 = \{1, 2\}$, $E_2 = \{1, 3, 4, 5\}$, $E_3 = \{3, 4\}$ et $E_4 = \{3, 4, 5\}$. On peut remarquer facilement sur la figure 4.3 que $\{E_1, E_2\}$ est un recouvrement, que $\{E_1, E_3\}$ est un pavage et $\{E_1, E_4\}$ est une partition.

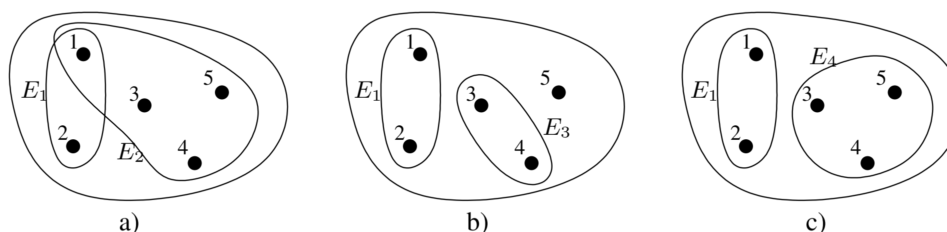



FIGURE 1.1 Illustration de a) Recouvrement, b) Pavage et c) Partition

Le *problème de recouvrement* (resp. *pavage*, *partition*) consiste à déterminer un recouvrement (resp. pavage, partition) dont la somme des poids des ensembles qui le forment est de poids minimum (resp. maximum, minimum/maximum).
On peut interpréter les contraintes du problème de recouvrement (resp. pavage, partition) comme le fait qu'un élément de $E$ doit être pris au moins une fois (resp. au plus une fois, exactement une fois).
Ces problèmes ont de multiples applications. Par exemple, considérons une région où l'on désire implanter des casernes de pompiers afin de couvrir toutes les villes. Pour chaque caserne potentielle, on détermine les communes desservies et le coût d'installation de la caserne. Comme l'on veut couvrir toutes les communes, il s'agit clairement d'un problème de recouvrement.

### Le problème du stable

Soient $G = (V, E)$ un graphe non-orienté et $c$ une fonction poids qui associe à tout sommet $v \in V$ un poids $c(v)$. Un *stable* de $G$ est un sous-ensemble $S$ de sommets de

$V$ tel qu'il n'existe aucune arête de $E$ entre 2 sommets de $S$. Le *problème du stable de poids maximum* consiste à déterminer un stable $S$ de $G$ tel que $c(S) = \sum_{v \in S} c(v)$ soit maximum.
La figure 1.2 donne par exemple un graphe où tous les sommets sont valués de poids 1. On peut noter que tous les sommets isolés sont des stables de poids 1 et qu'un stable a un poids au maximum 2, ce qui est obtenu par exemple par le stable $\{v_1, v_5\}$.

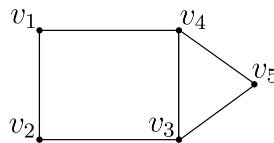



FIGURE 1.2 Illustration du problème du stable

### Le problème du voyageur de commerce

Soit $G = (V, E)$ un graphe non-orienté où les sommets $V$ sont appelées des villes et les arêtes entre deux villes des liaisons. Soit $c(e)$ un coût associé à la liaison $e$, pour toute $e \in E$. Le *problème du voyageur de commerce* (TSP, Traveller Salesman Problem) consiste à déterminer un cycle passant par chaque ville une et une seule fois. Un tel cycle est dit *hamiltonien*, on l'appelle aussi souvent *tour* ou *tournée*. Ce problème classique a trouvé de nombreuses applications dans des domaines pourtant aussi éloignés que la production de lait ou la construction de circuits intégrés.

### Le problème de coloration

Soit $S \subseteq I\!N$. Une *coloration* des sommets d'un graphe $G = (V, E)$ est une fonction $r : V \rightarrow S$ telle que $r(u) \neq r(v)$ pour tout couple de sommets adjacents $u, v$. Les éléments de l'ensemble $S$ sont appelés les *couleurs* disponibles. Une *$k$-coloration* est une coloration $c : V \rightarrow \{1, .., k\}$. Un graphe est dit *$k$-coloriable* s'il possède une $k$-coloration.
Tester si un graphe est $k$-coloriable est un problème NP-complet si $k \geq 3$, et il est polynomial si $k = 2$. En effet, si $k = 2$, il suffit de tester si le graphe est biparti, c'est-à-dire s'il ne contient pas des cycle impair. Ce qui peut se faire par un simple parcours de graphe. Soit $G = (V, E)$ un graphe non-orienté. Le *problème de coloration* consiste à déterminer le plus petit $k$ tel que $G$ soit $k$-coloriable.

# Chapitre 2

# Programmation Linéaire en Nombres Entiers

## 2.1 Définitions

Un *Programme Linéaire en Nombres Entiers* ou parfois *Programme Linéaire Mixte* (Integer Program ou Mixed Integer Program), noté MIP, est un problème d'optimisation sous contraintes et objectifs linéaires ($\mathcal{P}$) qui peut s'écrire de la façon suivante :

$$\begin{aligned} \text{Maximiser } & c_1^T x_1 + c_2^T x_2 \\ & \text{sous les contraintes} \\ & A_1 x_1 + A_2 x_2 \leq b \\ & x_1 \in \mathbb{R}^{n_1} \\ & x_2 \in \mathbb{Z}^{n_2}. \end{aligned}$$

où • $x$ est un vecteur appelé *variable*, ces $n$ composantes sont dites les *inconnues* du problème,
• la fonction $z : S \to \mathbb{R}$ est appelée *fonction objective* ou *objectif* (objective function), $= c_1, c_2$ sont des vecteurs et $A_1$ et $A_2$ des matrices avec $x_1$ partie continue de la solution et $x_2$ partie entière de la solution.
• les contraintes forment des inégalités qui sont appelées les *contraintes* (constraint) du problème.
• On désigne alors $x \in \mathbb{Z}^n$ comme étant la *contrainte d'intégrité* (ou d'entiéreté en Belgique ou d'intégralité au Québec) (integrity or integrality constraint).

On peut remarquer qu'un MIP peut être une maximisation ou une minimisation (il suffit de poser la fonction $z' = -z$. On appelle *inégalités* une contrainte $ax \leq b$ ou $ax \geq b$ : en cas de présences des deux contraintes on parle alors d'égalité $ax = b$.

Un vecteur $\bar{x}$ vérifiant les contraintes d'un MIP est dit *solution* ou *solution réalisable* du MIP. L'ensemble des solutions d'un MPI forme un *domaine de définition*. Le domaine de définition d'un MIP peut être : vide (dans ce cas, le problème n'admet pas de solutions), dans le cas contraire, le MIP admet des solutions. Sous certaines conditions, il peut exister des solutons $x^*$ dites *optimales*, c'est-à-dire qui maximisent la fonction $f(x)$ sur toutes les solutions du MIP.

On appelle *relaxation* le fait de “relâcher”, c'est-à-dire supprimer une contrainte du problème. Ainsi, un programme relaxé désignera un programme où l'on aura supprimé une ou plusieurs contraintes.
On appelle *relaxation continue* le fait de “relâcher” les contraintes d'intégrité du problème. Par abus de langage, on appelle aussi souvent *relaxation continue* le fait de résoudre le programme que où l'on a relâché les contraintes d'intégrité (l'expression désigne même parfois la solution optimale obtenue).

## 2.2 Difficulté de la PLNE

La première remarque qui peut sauter aux yeux est d'imaginer que résoudre un MIP revient à “arrondir” la solution de sa relaxation continue. L'exemple suivant témoigne de l'insuffisance de cette remarque :
Prenons un MIP à deux variables et une seule contrainte où toutes les fonctions sont linéaires, ce qui constitue le cas le plus simple que l'on puisse imaginer.

$$\begin{array}{l}\text{Maximiser } 10x_1 + 11x_2 \\ \quad 10x_1 + 12x_2 \leq 59 \\ \quad x_1 \text{ et } x_2 \geq 0 \\ \quad x_1, x_2 \text{ entiers.}\end{array}$$

En dessinant le domaine de définition, on obtient la figure 2.1 suivante : On remarque alors que l'optimum de la relaxation continue a une valeur objective de 59 et celui de l'optimum entier est de 54 seulement. Mais surtout, on peut noter l'écart complet de structure et de position des deux points optimum (qui sont ici chacun solution optimale unique).

En fait, résoudre un MIP donné est au moins aussi difficile que résoudre sa relaxation continue. En général, il l'est bien davantage : certains problèmes, comme c'est le cas lorsque toutes les fonctions sont linéaires, ont des relaxations continues polynomiales et des versions discrètes NP-difficiles. Inversement, si l'on ne sait pas résoudre efficacement la version continue d'un programme, il est quasi impensable de résoudre sa version discrète.

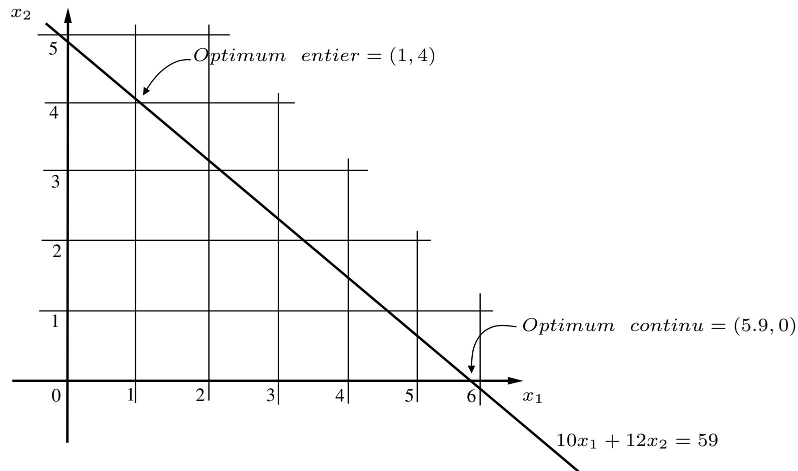



FIGURE 2.1 Ecart entre optimum entier et continu

## 2.3 Relaxation continue et solveurs

On s'intéresse ici à la résolution ainsi qu'à celle de la relaxation continue d'un MIP, en s'intéressant aux capacités des logiciels existants, appelés *solveurs*.

Un problème linéaire continu peut être résolu en temps polynomial (Khachiyan 1979). Il existe des algorithmes polynomiaux efficaces pour résoudre un programme linéaire comme ceux dits de *points intérieurs* initiés par Karmarkar (1984). Néanmoins l'algorithme du *simplexe* (Dantzig 1947) est le plus célèbre (et le plus efficace dans le cas général) des algorithmes de résolution, bien qu'il ne soit pas polynomial !
L'algorithme du simplexe repose sur le fait qu'une solution optimale d'un programme linéaire peut être prise parmi les sommets du polyèdre de $\mathbb{R}^n$ déterminé par $Ax \leq b$.

En revanche, la PLNE est un problème NP-difficile. Il est facile de montrer que la PLNE est un problème NP-difficile car de nombreux problèmes NP-difficiles peuvent être exprimés comme des PLNE.

Il existe de nombreux solveurs de PL : des solveurs commerciaux Cplex (IBM), Xpress, Gurobi (microsoft), et même Matlab ou Excel... ; des solveurs académiques Lp de COIN-OR, Soplex de la ZIB ; et des solveurs libres comme Glpk (gnu). Les meilleurs d'entre eux peuvent résoudre des PL jusqu'à 200000 variables et 200000 contraintes en quelques secondes.
En revanche, les solveurs entiers performants sont beaucoup moins performants : ils sont en général liés aux solveurs PL : Glpk par exemple ne dépassent pas quelques

100 aine de variables et contraintes ; les solveurs commerciaux Cplex ou Gurobi sont les plus performants (Xpress est un peu en-dessous) pouvant réussir parfois quelques milliers de variables/contraintes ; un solveur “universitaire” les rattrape : SCIP de la ZIB. Un des objectifs de ce cours est de comprendre comment et dans quels cas ces solveurs atteignent de telles capacités.

## 2.4 Modélisation

### 2.4.1 Un puissant outil de modélisation

Tout au long de ce cours, nous verrons à quel point l'écriture sous forme d'un MIP est un puissant outil de modélisation. En fait, on peut le voir souvent comme l'écriture naturelle algébrique d'un problème.

Inversement, comme nous l'avons cité, il n'existe pas de méthodes génériques efficaces pour résoudre un MIP. Le fait de ramener aussi facilement un problème à un MIP est donc parfois dangereux : on voit souvent des chercheurs ou des ingénieurs R&D conclure un travail de modélisation en affirmant que “comme on a ramené notre problème à un MIP que les solveurs n'arrivent pas à résoudre, notre problème est difficile et nous allons utiliser des méthodes approchées”. Cet état d'esprit, fort répandu malheureusement, est doublement faux. Tout d'abord, ramener un problème à un MIP ne prouve en rien la difficulté d'un problème : ce n'est pas une preuve de complexité, il peut ainsi exister d'autres pistes théoriques ou algorithmiques pour résoudre le problème d'origine. D'autre part, il peut exister plusieurs MIP modélisant le problème et différentes techniques pour le résoudre : c'est justement cette étude que nous allons mener dans cette section.

### 2.4.2 Modélisation par un PLNE

Cette section donne des pistes d'idées pour modéliser certains liens logiques entre des variables ou des contraintes du problème.

• *Cas simples*
Soient $a$ ,$b$ et $c$ des évênements correspondant aux variables de décisions binaires $x_a$, $x_b$ et $x_c$.

- Si $a$ et $b$ ne peuvent pas se produire tous les deux : $x_a + x_b \leq 1$.
- Si $a$ se produit alors $b$ doit se produire : $x_a \leq x_b$.
- On peut noter que l'inégalité précédente modélise heureusement la contraposée de la proposition logique correspondante : si $b$ ne se produit pas, $a$ ne doit pas

se produire.

- Si $a$ se produit, alors $b$ et/ou $c$ doivent se produire : $x_a \leq x_b + x_c$.
- Si $a$ ne se produit pas, alors une quantité $y$ doit être nulle, sinon $y$ est libre dans $I\!R$. On doit fixer une quantité $M$ telle que $y$ ne peut jamais être supérieure à $M$ lorsqu'on atteint l'optimum du problème. Une telle constante $M$ existe, car sinon le problème serait non borné : $y \leq Mx_a$ (contrainte dite de "big M").

• *Maximiser la valeur minimale d'un ensemble de fonctions linéaires*
Si on veut maximiser la plus petite valeur prise par un ensemble de fonctions linéaires $a^i x$, $i = 1, ...m$, il suffit d'ajouter une variable $z$ et les contraintes $z \leq a^i x$, $i = 1, ...m$ : la fonction objective devient alors Max $z$.

• *$k$ contraintes parmi $n$*
On dispose d'un lot de $n$ contraintes $a^1 x \leq b^1, a^2 x \leq b^2, ..., a^n x \leq b^n$ parmi lesquelles $k$ au moins doivent être satisfaites (c'est-à-dire que les autres sont libres d'être satisfaites ou non). Pour chacune des contraintes $a^i x \leq b^i$, $i = 1, ..., n$, on détermine une valeur $M_i$ suffisamment grande pour que $a^i x \leq b^i + M_i$ soit satisfaite quelque soit $x$.
On pose alors $y_1, ..., y_n$ des variables binaires et ce cas de figure se modélise alors de la façon suivante.

$$
\begin{array}{lcl}
a^1 x \leq b^1 & & a^1 x \leq b^1 + M_1 y_1 \\
a^2 x \leq b^2 & & a^2 x \leq b^2 + M_2 y_2 \\
... & \Rightarrow & ... \\
a^n x \leq b^n & & a^n x \leq b^n + M_n y_n \\
 & & \sum_{i=1}^{n} y_i = n - k
\end{array}
$$

Remarquons que si une seule contrainte doit être satisfaite parmi deux (c'est-à-dire si $k = 1$ et $n = 2$), on peut utiliser une seule variable binaire $y$ en posant $y_1 = y$ et $y_2 = 1 - y$.

• *Implication entre contraintes*
Soit $a^1 x \leq b^1$ et $a^2 x \leq b^2$ deux contraintes telles que si $a^1 x < b^1$, alors $a^2 x \leq b^2$ doit être satisfaite, mais que, par contre, si $a^1 x \geq b^1$, alors $a^2 x \leq b^2$ peut ou non être satisfaite.
On prend $M$ tel que $-a^1 x \leq -b^1 + M$ et $a^2 x \leq b^2 + M$ soient vérifiées pour toute valeurs de $x$. On peut utiliser une variable de décision binaire $y$ et écrire alors :

$$-a^1 x \leq -b^1 + M(1 - y) \quad (2.1)$$
$$a^2 x \leq b^2 + My \quad (2.2)$$

En effet, si $a^1 < b$, alors la contrainte (2.1) implique que $y = 0$, et ainsi la contrainte (2.2) est équivalente à $a^2 x \leq b^2$ qui doit donc être satisfaite. Pour le cas contraire (i.e. si $a^1 x \geq b^1$), alors $y$ peut prendre la valeur 0 ou 1, c'est-à-dire que $a^2 x \leq b^2$ peut être satisfaite ou non.

### 2.4.3 Linéarisation

On appelle *linérisation* l'ensemble des astuces permettant de transformer en un programme mathématique qui aurait un objectif ou des contraintes non linéaires à un MIP. Généralement, ces transformations demandent l'ajout de nombreuses variables supplémentaires.

Cette section comporte quelques astuces pour effectuer une telle linéarisation à partir d'une forme quadratique De manière plus générale, dans le cadre des problèmes d'optimisation combinatoire, la section suivante tente de montrer comment modéliser directement un problème en utilisant un PLNE.

• *Une variable à valeurs dans un espace discret*
Soit $x$ une variable prenant ses valeurs parmi les $n$ possibilités $v_1, ..., v_n \in \mathbb{R}$. On peut alors poser $n$ variables de décisions binaires $y_1, ...., y_n$ telle que $y_i = 1$ si $x = v_i$ et 0 sinon.
Ce cas peut alors se modéliser par les deux contraintes $x = \sum_{i=1}^{n} v_i y_i$ et $\sum_{i=1}^{n} y_i = 1$.

• *Ecriture en variables binaires*
Si l'on peut déterminer des bornes sur les variables, on peut utiliser l'idée du cas précédent pour écrire un PLNE en variables entières comme un PLNE à variables binaires. En effet, considérons une variable $x$ à valeurs entières entre 0 et $u$, $u \in \mathbb{N}$. Soit $n$ tel que $2^n \leq u < 2^{n+1}$. On remplace alors $x$ par sa représentation binaire : $x = \sum_{i=1}^{n} 2^i y^i$ où $y_i \in \{0, 1\}$ pour $i = 1, ..., n$.

• Le "ou" numérique
On veut représenter une variable $x$ devant prendre des valeurs soit 0, soit être plus grande que $L$ où $L$ et $x$ sont bornées par une valeur $M$.
On ajoute une variable $y \in \{0, 1\}$ et on utilise les contraintes

$$x \geq Ly \qquad \text{et} \qquad x \leq My.$$

• Carré d'une variable binaire
Soit $x \in \{0,1\}$. Alors la variable $x^2$ est équivalente à la variable $x$.

• Produit de deux variables binaires :
Soit $x \in \{0,1\}$ et $y \in \{0,1\}$, on veut obtenir une variable $e$ ayant la valeur $e = xy$.
En fait, on a le résultat suivant :

$$e = xy \Leftrightarrow \begin{cases} e \leq x \\ e \leq y \\ e \geq x + y - 1 \\ e \geq 0 \\ e \in I\!R \end{cases}$$

On peut généraliser ce résultat au produit d'une variable binaire par une variable entière (bornée), au produit de plusieurs variables binaires, au carré d'une variables binaires,...

Ainsi, au total de ces trois remarques, on peut remarquer que toute forme quadratique peut se ramener à un PLNE binaire ! Mais cela se fait au prix fort, en ajoutant de nombreuses variables et contraintes.

## 2.5 Formulations classiques

### 2.5.1 Le problème du sac-à-dos

La formulation PLNE du problème de sac-à-dos est très simple. On utilise pour chaque objet $i \in \{1, ..., n\}$, une variable entière $x_i$ correspondant au nombre de fois où l'objet $i$ est choisi. Le problème du sac-à-dos est donc équivalent au programme en nombres entiers suivant.

$$\begin{aligned} \text{Max } & \sum_{i=1}^{n} c_i x_i \\ & \sum_{i=1}^{n} a_i x_i \leq b, \qquad (2.3) \\ & p_i \leq x_i \leq q_i, \text{ pour } i = 1, ..., n, \\ & x_i \in I\!N, \text{ pour } i = 1, ..., n. \end{aligned}$$

La contrainte (2.3) est dite *contrainte de sac-à-dos*. Elle est l'unique contrainte de ce problème qui est pourtant NP-complet.

### 2.5.2 Recouvrement, pavage et partition

Ces problèmes peuvent se modéliser en utilisant des variables binaires $x_1, ..., x_m$ associées aux sous-ensembles $E_1, ..., E_m$. Pour cela, on considère $A$ la matrice en 0-1 dont les lignes correspondent aux éléments $1, ..., n$ et les colonnes aux sous-ensembles $E_1, ..., E_m$ et dont les coefficients sont $A_{ij} = 1$ si $i \in E_j$ et 0 sinon. Ainsi les trois problèmes peuvent s'écrire.

| Recouvrement | Pavage | Partition |
|---|---|---|
| $\text{Min} \sum_{i=1}^{m} c_j x_j$ | $\text{Max} \sum_{i=1}^{m} c_j x_j$ | $\text{Max (ou Min)} \sum_{i=1}^{m} c_j x_j$ |
| $Ax \geq 1\!\!1$ | $Ax \leq 1\!\!1$ | $Ax = 1\!\!1$ |
| $x \in \{0,1\}^m$ | $x \in \{0,1\}^m$ | $x \in \{0,1\}^m$ |

où $1\!\!1$ est un vecteur dont chaque composante est 1.
Dans l'exemple associée à la figure 4.3, la matrice $A$ est alors

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}.$$

### 2.5.3 Le problème du stable

Notons $\chi^S$ le vecteur d'incidence d'un stable $S$ dans $V$, c'est-à-dire un vecteur indexé sur les sommets de $V$ tel que $\chi^S(u) = 1$ si $u \in S$ et 0 sinon. Un tel vecteur doit nécessairement vérifier, pour toute arête $uv$ de $E$, qu'au plus une seule des deux extrémités parmi $u$ et $v$ peut être prise dans $S$. Ce que l'on peut décrire par l'inégalité linéaire $x(u) + x(v) \leq 1$, appelée *inégalité aux arêtes.*
Inversement, considérons un vecteur binaire $x^*$, indexé sur les sommets de $V$, qui vérifie toutes les inégalités associées aux arêtes du graphe $G$. Alors $x^*$ est un vecteur d'incidence d'un stable de $G$. En effet, si l'on pose $W$ l'ensemble des sommets $v \in V$ tel que $x^*(v) = 1$, alors clairement $W$ ne contient pas deux sommets adjacents.
Ainsi le problème du stable est équivalent au PLNE suivant :

$$\begin{aligned} \text{Max} \sum_{u \in V} c(u)x(u) & \\ x(u) + x(v) \leq 1, & \quad \text{pour tout } uv \in E, \\ x(u) \in \{0,1\}, & \quad \text{pour tout } u \in V. \end{aligned} \tag{2.4}$$

On nomme *compacte* une formulation qui contient un nombre polynomial de variables et de contraintes. On peut noter que cette formulation est compacte car elle contient un nombre polynomial de contraintes et de variables en fonction de la taille de l'entrée du problème. En effet, il y a une variable par sommet du graphe et une inégalité par arête. On appelle d'ailleurs cette formulation du problème du stable "formulation aux arêtes".
Le problème du stable est donc équivalent à un PLNE qui possède un nombre réduit de contraintes et de variables. Le problème est pourtant NP-complet et difficile à résoudre.

D'autres formulations de ce problème sont possibles. Par exemple, on peut remplacer les contraintes (2.4) par les contraintes dites *de cliques*

$$\sum_{u \in K} x(u) \leq 1, \text{ pour toute clique } K \text{ de } G. \quad (2.5)$$

Les contraintes d'arêtes (2.4) sont contenues dans la famille des contraintes de cliques (2.5). On peut constater que la nouvelle formulation est aussi équivalent au problème du stable mais elle contient d'avantage de contraintes (en fait un nombre exponentiel dans le pire des cas). En revanche, la solution de la relaxation linéaire de la formulation par les cliques sera plus proche de la solution optimale du problème du stable que la relaxation de la formulation par les arêtes. Or, les algorithmes par séparation/évaluation nous disent que plus la relaxation est bonne, plus l'algorithme est rapide.

On peut se poser plusieurs questions : Comment choisir une bonne formulation ? Comment décider si une contrainte est utile ou non à la relaxation ?

### 2.5.4 Le problème du voyageur de commerce

Regardons le problème assymétrique : c'est-à-dire que les coûts des arcs $c_{ij} \neq c_{ji}$ pour tout arête $ij \in e$. On considère le graphe complet $D = (V, A)$, avec des coûts sur les arcs $c_{ij}$. On recherche le cycle orienté, appelé *tour* qui contient les $n$ villes et qui soit de taille minimale. On définit les variables $x_{ij} = 1$ si l'arc $(i, j)$ est dans le tour et 0 sinon. Même avec ce formalisme déjà très précis, il existe plusieurs formulations possibles.

Regardons le PLNE suivant.

$$\begin{array}{lll}
\text{Min} \displaystyle\sum_{i,j} c_{ij} x_{ij} & & \\
\sum_{j \in V} x_{ij} = 1 & \text{pour tout } i \in V, & (2.6) \\
\sum_{i \in V} x_{ij} = 1 & \text{pour tout } j \in V, & (2.7) \\
0 \leq x_{ij} \leq 1 & \text{pour tout } (i,j) \in V \times V, & \\
x_{ij} \in \mathbb{N} & \text{pour tout } (i,j) \in V \times V. &
\end{array}$$

La relaxation linéaire de ce PLNE est entière car la matrice est totalement unimodulaire (voir chapitre suivant). Malheureusement, les solutions peuvent contenir plusieurs cycles orientées, que l'on appelle *sous-tours*. Les contraintes 2.6 sont appelées *inégalités d'affectation*.

•*"Elimination des sous-tours" par la formulation MTZ*

Pour éliminer les sous-tours, on peut aussi utiliser la formulation de Miller-Tucker-Zemlin (MTZ). On ajoute de nouvelles variables réelles $u_i$, $i = 1, ..., n$, associées aux villes et les contraintes :

$$u_1 = 1, \tag{2.8}$$
$$2 \leq u_i \leq n \text{ pour tout } i \neq 1, \tag{2.9}$$
$$u_i - u_j + 1 \leq n(1 - x_{ij}) \text{ pour tout } i \neq 1, j \neq 1. \tag{2.10}$$

On apelle ces dernière inégalités *inégalités MTZ*. Elles permettent d'éliminer les sous-tours :
- car, pour tout arc $(i, j)$ où $x_{ij} = 1$, elles forcent $u_j \leq u_i + 1$,
- si une solution du PLNE formé pas les inégalités (2.6) contient plus d'un sous-tour, alors l'un deux au moins ne contient pas le sommet 1 et, sur ce sous-tour, les variables $u_i$ s'incrémentent à l'infini.

On peut par contre remarquer que, de surcroit, dans le cas d'une solution réalisable pour le PLNE formé des inégalités (2.6) et des 3 inégalités MTZ, les variables $u_i$, $i = 1, ..., n$ indiquent la position de la ville $i$ dans le tour. Ainsi, si on veut indiquer qu'une ville $i$ doit être positionnée avant une autre, ou proche d'une autre ville $j$, on peut ajouter des contraintes sur concernant $u_i$ et $u_j$.

Cette formulation possède $n$ variables de plus, mais elle a l'énorme avantage d'être compacte. On peut donc directement utiliser une procédure de branchement et séparation (Branch& Bound) et donc un solveur entier. Cette formulation est plus faible que la formulation par les coupes et elle n'a pas d'intérêt autre qu'être compacte.

•*"Elimination des sous-tours" en brisant les sous-tours*

Pour éliminer les sous-tours, on peut ajouter les contraintes suivantes, dites contraintes de sous-tours :

$$\sum_{i \in S, j \in S} x_{ij} \leq |S| - 1, \text{ pour tout } S \subset V, |S| > 1, S \neq V \tag{2.11}$$

Ces contraintes empèchent les solutions d'avoir des sous-tours. Néanmoins, ces contraintes sont en nombres exponentielles. Pour pouvoir résoudre une telle formulation, on utilise les algorithmes de coupes, qui mettent en pratique les résultats obtenus lors d'une étude polyédrale du problème. Le principe consiste à ajouter progressivement les inégalités dans la formulation. Ici l'ajout se fait par un algorithme, dit algorithme de séparation, qui revient à détecter des cycles dans un graphe.

•*"Elimination des sous-tours" par la connexité par les coupes*

Une autre façon d'interdire les sous-tours est de remarquer qu'une solution du TSP est un ensemble d'arcss $C$ qui forme un sous-graphe partiel connexe, c'est-dire qu'il y a au moins un arc sortant de chaque ensemble de sommets, c'est-à-dire au moins un sommet par coupes du graphe.

$$\sum_{e \in \delta^+(W)} x(e) \geq 1, \text{ pour tout } W \subsetneq V et W \neq \emptyset, \tag{2.12}$$
$$\sum_{e \in \delta^-(W)} x(e) \geq 1, \text{ pour tout } W \subsetneq V et W \neq \emptyset,$$

Les contraintes *de coupes* (2.20) quant à elles, obligent l'ensemble d'arêtes $C = \{e \in E \mid x(e) = 1\}$ à former un graphe connexe. En effet, si ce graphe n'était pas connexe, alors il existerait une coupe dans le graphe qui n'intersecterait pas $C$.
On peut montrer qu'en fait les contraintes (2.11) et (2.20) sont équivalentes (cela se montre en utilisant les contraintes de degré).
Il y a à nouveau un nombre exponentiel de contraintres de connexité. On utilisera alors un algorithme de coupes et branchement. C'est actuellement le principe qui a permis d'obtenir les meilleures solutions.

•*"Cas symétrique"*

Pour le cas symétrique du TSP, on peut simplifier les formulations précédentes, on obtient alors :
Soit $x$ une variable associée aux liaisons de $E$ telle que $x(e) = 1$ si $e$ prise dans la solution et 0 sinon. Le problème du voyageur de commerce est équivalent au PLNE suivant.

$$Min \sum_{e \in E} c(e)x(e)$$

$$\sum_{e \in \delta(v)} x(e) = 2, \text{ pour tout } u \in V, \tag{2.13}$$

$$\sum_{e \in \delta(W)} x(e) \geq 2, \text{ pour tout } W \subsetneq V \text{ et } W \neq \emptyset, \tag{2.14}$$

$$x(e) \in \{0,1\}, \text{ pour tout } e \in E.$$

En fait, les contraintes (2.19) dites *de degré* forcent à 2 le nombre d'arêtes incidentes à un même sommet. Les contraintes *de coupes* (2.14) quant à elles, obligent l'ensemble d'arêtes $C = \{e \in E \mid x(e) = 1\}$ à former un graphe connexe. En effet, si ce graphe n'était pas connexe, alors il existerait une coupe dans le graphe qui n'intersecterait pas l'ensemble $C$ or ceci est impossible car cette contrainte est exigée pour chaque coupe. On peut remarquer qu'il y a un nombre exponentielle de contraintes de connexité. De plus, leur énumération est loin d'être évidente. On peut donc se poser la question : Comment résoudre un programme linéaire possédant un nombre exponentiel de contraintes ? Comment produire les contraintes d'un tel programme linéaire ?

### 2.5.5 Le problème de coloration

Soit un graphe $G = (V, E)$. Une formulation en variables entières peut être donnée de la façon suivante. On associe à chaque sommet $u$ de $V$ un vecteur binaire à $K$ dimensions $x_u = (x_u^1, ..., x_u^K)$, où $K$ est une borne supérieure sur la coloration de $G$ (au maximum $K = |V|$). Comme l'on désire minimiser le nombre de couleur, on ajouter une variable binaire $w_l$ par couleur $l = 1, ..., K$ indiquant si cette couleur a été utilisée ou non. Le problème est donc équivalent au programme

$$\text{Min } \sum_{l=1}^{K} w_l$$

$$\sum_{l=1}^{K} x_u^l = 1, \qquad \text{pour tout } u \in V, \tag{2.15}$$

$$x_u^l + x_v^l \leq w_l, \qquad \text{pour tout } e = uv \in E \text{ et } 1 \leq l \leq K, \tag{2.16}$$

$$x_u^l \in \{0,1\}, \qquad \text{pour tout } u \in V \text{ et } 1 \leq l \leq K.$$

La contrainte (2.15) oblige chaque sommet à être affecté à une et une seule couleur. La contrainte (2.16) oblige que deux sommets adjacents n'aient pas la même couleur.

Dans cette formulation, il y a dans le pire des cas $n^2 + n$ variables binaires et $n * m$ contraintes. Malgré sa taille polynomiale raisonnable, cette formulation s'avère être en pratique très difficile à résoudre. (En fait, on peut remarquer que ce PLNE peut être vu comme un programme linéaire modélisant le problème du stable dans un graphe particulier constitué de $k$ copies du graphes d'origine où chaque copie d'un sommet est reliée à toutes les autres copies de ce même sommet).

Cette formulation, en pratique, s'avère très difficile à résoudre par les algorithmes de branchements. Cependant, d'autres formulations ont été proposées, plus pour donner des bornes inférieures au problème qu'un réel résultat exact optimal. La formulation suivante s'est avérée par exemple très performante pour donner des bornes inférieures. Elle repose sur le fait qu'une coloration est en fait une couverture d'un graphe par un nombre minimum de stables.
Soit $\mathcal{S}$ l'ensemble des stables non vides de $G$. On associe à chaque stable $S \in \mathcal{S}$ une variable binaire $t_S$. Le problème de coloration est alors équivalent au programme en nombres entiers suivant (Mehrotra et Trick 1995).

$$\text{Min} \sum_{S \in \mathcal{S}} t_S$$

$$\sum_{S \in \mathcal{S} \mid u \in S} t_S = 1, \quad \text{pour tout } u \in V, \tag{2.17}$$

$$t_S \in \{0, 1\}, \quad \text{pour tout } S \in \mathcal{S}. \tag{2.18}$$

En fait cette formulation possède la particularité de contenir un nombre exponentiel de variables et un nombre très réduit de contraintes. L'algorithme nécessaire pour résoudre une telle formulation s'appelle *algorithme de génération de colonnes* car ils consistent à générer des variables que l'on ajoute itérativement au problème. Ce procédé peut être vu comme le procédé dual des méthodes de coupes auxquelles ce cours s'intéresse.

## 2.6 Rappels théoriques sur la résolution

### 2.6.1 PLNE compact

Un PLNE compact peut directement être résolu par des méthodes génériques.
Ces méthodes reposent essentiellement sur le principe de branchements qui permet d'énumérer intelligemment toutes les solutions du problème en évitant de les explorer toutes.
Les solveurs les plus puissants utilisent d'autres méthodes qui s'ajoutent à celles de branchement :
- prétraitement : une analyse approfondie de la structure des variables et des contraintes

permet d'éliminer certaines variables et contraintes. Il est parfois possible de réduire un MIP de 50% ou plus !
- fixation : en déterminant des implications logiques entre variable, les algorithmes peuvent se limiter à faire varier seulement certaines d'entre elles lors d'une recherche.
- ajouts de contraintes : il est possible d'ajouter des contraintes qui vont servir à améliorer les valeurs de relaxation continue et de couper des solutions lors de l'exploration.
Ces techniques proviennent de l'approche polyédrale et ont permis d'améliorer énormément les solveurs MIP.

Néanmoins, alors que la PL peut être dite aujourd'hui facile à résoudre, les PLNE restent difficiles à résoudre. Il n'est pas rare de devoir attendre plusieurs minutre pour résoudre un PLNE de quelques dizaines de variables/contraintes. Et s'il s'agit de centaines, les solveurs ne savent même parfois pas le résoudre.

### 2.6.2 Algorithme de coupes et branchements

Dans la partie suivante, nous étudierons le cadre théorique générale de l'ajout de contraintes à un PLNE qui se nomme "approches polyédrales". Néanmoins, nous avons déjà rencontré deux cas où on considère un grand nombre de contraintes :
- les formulations PLNE contenant un grand nombre de contraintes (potentiellement exponentiel) que l'on ne peut donc pas énumérer.
- des exemples où l'ajout de contraintes supplémentaires permet d'obtenir une meilleure valeur de relaxation : on appelle cela parfois le "renforcement" de formulation.

Nous tenterons de répondre plus tardivement aux question : quelle contrainte ajoutée pour améliorer une relaxation ? quelle formulation considérer etc ?

Dans ce chapitre, nous allons étudier comment gérer algorithmiquement l'ajout d'un grand nombre de contraintes dans un PLNE.

#### Coupes et séparation

Nous considérons ici un programme linéaire

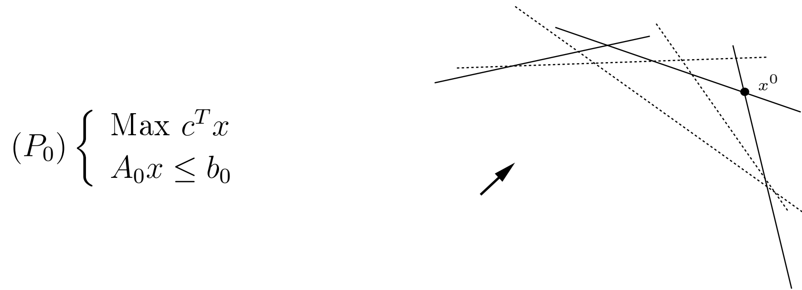
$$(P) \quad Max\{c^T x \mid Ax \leq b\}$$

comportant $n$ variables mais où $A$ contient un nombre important, par exemple exponentiel par rapport à $n$ de contraintes (on dit que le programme est non compact sur les contraintes.
Un tel programme ne peut donc pas être introduit comme instance d'un algorithme de B&B et donc ne peut pas être utilisé directement dans les solveurs entiers. Nous allons

pourtant voir qu'il existe un cadre algorithme efficace, les algorithmes de coupes pour ces programmes.

Choisissons tout d'abord un sous-ensemble de $A_0x \leq b_0$ de contraintes de $Ax \leq b$ tel qu'il existe une solution finie $x^0$ au problème $(P_0) \quad Max\{c^Tx \mid A_0x \leq b_0\}$. La solution $x_0$ est donc un point extrême du polyèdre défini par le système $A_0x \leq b_0$. On notera alors $(A \setminus A_0)x \leq (b \setminus b_0)$ les inégalités de $Ax \leq b$ qui ne sont pas dans $A_0x \leq b_0$.

La figure suivante représente en 2 dimensions, les contraintes de $A_0x \leq b_0$ en traits pleins et celles de $(A \setminus A_0)x \leq (b \setminus b_0)$ sont en pointillés. On notera alors $(A \setminus A_0)x \leq (b \setminus b_0)$ ces inégalités. La flèche indique la "direction d'optimisation" c'est-à-dire la direction orthogonale à toute droite d'équation $z = c^Tx$.

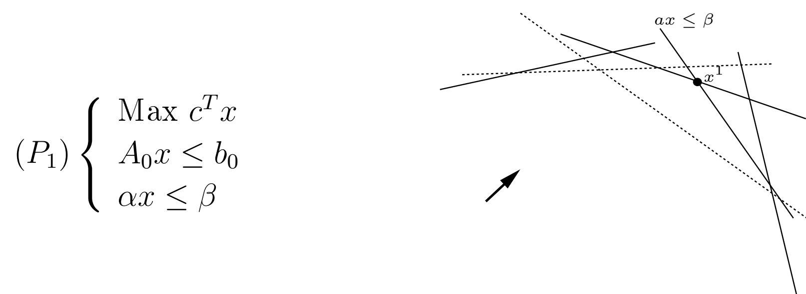



On peut remarquer que $x^0$ n'est pas pas forcément un point extrême de $Ax \leq b$ : cela se produit s'il existe une inégalité de $(A \setminus A_0)x \leq (b \setminus b_0)$ qui n'est pas satisfaite par $x^0$ : on dit que $x^0$ *viole* la contrainte. Pourtant $x^0$ peut être un point extrême de $Ax \leq b$ : dans ce cas, $x^0$ serait alors une solution optimale de $(P)$ ! Cette idée donne lieu au problème de séparation.

**Problème de séparation** Etant donné un point $x \in I\!R^n$, le *problème de séparation* associé à $Ax \leq b$ et $x$ consiste à déterminer si $x$ satisfait toutes les inégalités de $Ax \leq b$ et sinon à trouver une inégalité de $Ax \leq b$ violée par $x$.

On peut répondre à ce problème par un algorithme que l'on appelle *algorithme de séparation*. Si on connaît un tel algorithme de séparation, on sait donc déterminer une inégalité $ax \leq \beta$ qui est violée par le point extrême $x^0$. On ajoute alors cette contrainte aux inégalités du système courant : on appelle alors une telle inégalité une *coupe* car elle "coupe" le point indésirable de l'espace des solutions possibles.

Sur la figure suivante, la contrainte $\alpha x \leq \beta$ qui était violée par $x^0$ a été ajoutée au système courant : on voit qu'elle a "séparé" une partie de l'espace indésirable. En posant alors $A_1x \leq b_1 = (A_0 \leq b_0) \cup (ax \leq \beta)$ et $(P_1) \quad Max\{c^Tx \mid A_1x \leq b_1\}$, on




peut reproduire le même algorithme de séparation pour le nouveau point extrême $x^1$ obtenue par la résolution optimale du PL $(P1)$.

En réitérant l'algorithme de séparation autant tant que l'on peut ajouter des contraintes violées, on obtient alors une algorithme appelé *méthode de coupes* (cutting plane algorithm). A la fin d'une méthode de coupes, on obtient donc nécessairement un point extrême $x^*$ qui est point extrême du système courant et qui n'est pas violé par une contrainte de $Ax \leq b$ : on obtient donc une solution optimale de $(P)$.

La question importante est donc alors de connaître la terminaison et la complexité d'une méthode de coupes. Ceci nous est donné par le résultat extrêmement puissant suivant :

**Théorème 2.6.1** [ Grötschel, Lovász et Schrijver [?] (1981)]
*Une méthode de coupes sur un système $Ax \leq b$ de contraintes est polynomial si et seulement si l'algorithme de séparation associé à $Ax \leq b$ est polynomial.*

Ce résultat fondamental permet ainsi de manipuler des formulations exponentielles pour la PLNE ! Il indique ainsi qu'optimiser est équivalent à séparer.

Une autre conséquence concerne la taille du système linéaire utilisé : en fait, comme on ajoute une inégalité un nombre polynomial de fois : le système courant a au plus une taille polynomiale ! Ainsi, au lieu de manipuler un système exponentiel, un simple système polynomial est suffisant.
Cette remarque n'est pas surprenant, en effet, pour définir un point extrême (solution) il suffit de $n$ contraintes (linéairement indépendantes) satisfaites à l'égalité : le reste peut être donc être "écarté" du PL correspondant : la question est alors : comment trouver ces $n$ contraintes ? En effet, si on les a, une seule utilisation de l'algorithme de séparation permettrait de tester si le point est une solution de $Ax \leq b$.

### Exemples de séparation de contraintes

Dans cette section, nous aborderons des exemples d'algorithmes de séparation de contraintes célèbres.
• *Contraintes de coupes pour le TSP*
Considérons la formulation suivante pour le voyageur de commerce symétrique que nous avons vu dans le début de cette partie.

$$
\begin{aligned}
& Min \sum_{e \in E} c(e)x(e) \\
& \sum_{e \in \delta(v)} x(e) = 2, \text{ pour tout } u \in V, && (2.19)\\
& \sum_{e \in \delta(W)} x(e) \geq 2, \text{ pour tout } W \subsetneq V \text{ et } W \neq \emptyset, && (2.20)\\
& x(e) \geq 0, \text{ pour tout } e \in E \\
& x(e) \in \{0,1\}, \text{ pour tout } e \in E.
\end{aligned}
$$

Les contraintes (2.20) sont en nombre exponentiel. On peut construire un algorithme de coupes en prenant comme ensemble initial les contraintes de degré et les contraintes triviales.
Le but ici est de résoudre la relaxation linéaire de cette formulation PLNE. Notons $x^*$ une solution optimale du PL limité aux contraintes initiales. Le problème de séparation peut alors s'écrire de la façon suivante : déterminer une contrainte (2.20) violée par $x^*$ si elle existe et dire sinon qu'il n'en n'existe pas.
Ce problème se ramène en fait à déterminer une coupe min dans le graphe $G$ en utilisant pour capacité le vecteur $x^*$ associée aux arêtes de $G$. En effet, si l'on dispose d'une coupe $\delta(W^*)$ avec $W^* \subsetneq V$ et $W^* \neq \emptyset$ et $x^*(\delta(W^*)) = \sum_{e \in \delta(W)} x^*(e)$ minimum, on a deux cas :
- soit $x^*(\delta(W^*)) < 2$ dans ce cas, on a déterminé une contrainte $\sum\limits_{e \in \delta(W^*)} x(e) \geq 2$ violée par $x^*$
- soit $x^*(\delta(W)) \geq 2$ dans ce cas, toutes les contraintes (2.20) sont non violées.
Comme on sait déterminer une coupe-min avec des capacités positives (et $x^*$ est un vecteur positif) en temps polynomial, on sait donc résoudre en temps polynomial cette formulation exponentielle.
• *Contraintes de clique pour le stable*
On a vu qu'une formulation possible pour le problème du stable est d'utiliser les contraintes de cliques.

$$\begin{aligned}
\text{Max } & \sum_{u \in V} c(u)x(u) \\
& \sum_{u \in K} x(u) \leq 1, \text{ pour toute clique } K \text{ de } G. \\
& x(u) \in \{0,1\}, \text{ pour tout } u \in V.
\end{aligned} \tag{2.21}$$

Cette formulation contient un nombre exponentiel de cliques. On peut prouver que le problème de séparation associé à ces contraintes est équivalent à recherche une clique de plus grand poids positifs associés aux sommets dans le graphe $G$ : or ce problème est NP-complet. Ce qui signifie que l'on ne peut pas construire un algorithme de coupes pour cette formulation.

En revanche, on peut considérer la formulation suivante

$$\begin{aligned}
\text{Max } & \sum_{u \in V} c(u)x(u) \\
& x(u) + x(v) \leq 1, \text{ pour tout } uv \in E, \\
& \sum_{u \in K} x(u) \leq 1, \text{ pour toute clique } K \text{ de } G, |K| \geq 3. \\
& x(u) \in \{0,1\}, \text{ pour tout } u \in V.
\end{aligned} \tag{2.22}$$

En effet, seules les contraintes d'arêtes sont nécessaires à la formulation : elles peuvent donc être énumérées et utilisées tout au long de l'algorithme de coupes. Les contraintes associées à des cliques de tailles au moins 3 seront elles utilisées dans un algorithme de coupes heuristiques : c'est-à-dire qu'au lieu de rechercher s'il existe ou non une contrainte de cliques violées, on recherche heuristiquement une contrainte de cliques violée.
Cela peut être fait en utilisant par exemple un algorithme glouton : pour une solution $x^*$ du problème relaxée, on recherche un sommet de grand poids, puis on essaye d'ajouter un sommet de grand poids formant une clique avec le sommet précédent et on réitère l'idée. Cet algorithme glouton n'est évidemment pas exacte mais permet d'obtenir des contraintes de cliques en temps polynomial : on peut donc le réitérer dans un processus de génération de contraintes.
L'algorithme présenté précedemment ne permet de résoudre la relaxation linéaire du PLNE du stable donné précédemment : toutefois, on obtient à la fin une bien meilleure valeur de relaxation linéaire que la formulation limitée aux contraintes dites "aux arêtes".

### Branchements

Ainsi, dans le cas (rare) où l'on connaît un système linéaire définissant un polyèdre entier ET que l'on connait un algorithme de séparation polynomial pour chacune de ces contraintes, on obtient alors une méthode de coupes qui permet de résoudre un PLNE en temps polynomial.
Cependant, il y a peu d'espoir de connaître un tel système pour un problème d'Optimisation Combinatoire quelconque (surtout NP-complet). D'autre part, le problème de séparation sur certaines classes d'inégalités valides peut être lui-même NP-difficile. Dans ce cas, on ne peut disposer que de techniques de séparation approchées.
Ainsi, une méthode de coupes seule peut ne fournir que des solutions fractionnaires (non optimales). Dans ce cas, on exécute une étape de branchement qui peut consister à choisir une variable fractionnaire $x_i$ dans la solution et à considérer deux sous-problèmes du problème courant en fixant $x_i$ à 0 pour l'un et à 1 pour l'autre. On applique alors la méthode de coupes pour les deux sous-problèmes. La solution optimale du problème sera donc la meilleure entre les deux solutions entières obtenues pour les deux sous-problèmes. Cette phase de branchement est répétée de manière récursive jusqu'à l'obtention d'une solution entière optimale. Cette combinaison de la méthode de branchements et d'une méthode de coupes au niveau de chaque noeud de l'arbre de branchement est appelée *méthode de coupes et branchements* (*Branch and Cut method*).
Cette méthode s'est révélée très efficace pour la résolution de problèmes d'optimisation combinatoire réputés pour être difficiles tels que le problème du voyageur de commerce ou celui de la coupe maximum.

La mise en oeuvre d'un algorithme de B&C est donc assez complexe : elle nécessite de bien gérer un algo de B&B et plusieurs algo de séparation. Il faut également gérer les contraintes en très grand nombre et un solveur linéaire... Il est également possible, afin d'éviter de gérer trop de contraintes, de ne pas toute les considérer simultanément.
Il existe des "framework" souvent en langage C++ qui permettent de donner le cadre global d'un algorithme de B&C. Ces frameworks gèrent ainsi l'arbre de branchement, l'interfaçage avec un solveur linéaire, la gestion des contraintes, etc. Les solveurs sont principalement des outils libres, à part Concert Technology, produit lié à Cplex d'Ilog (mais qui n'est pas utilisable dans tous les cas, principalement s'il y a un grand nombre de contraintes) : on peut citer Abacus de la Zib (plus maintenu), BCP de Coin-Or et SCIP de la Zib.

# Chapitre 3

# Mise en œuvre : exemple du sous-graphe acyclique induit

kDans cette section, nous allons présenter comment utiliser un solveur afin de résoudre un PL ou un PLNE. Nous allons nous appuyer sur l'exemple du sous-graphe acyclique en regardant deux formulations : l'une compacte et l'autre non-compacte. La résolution d'un PLNE compacte présentée ici par l'utilisation d'un fichier au format universel, le format lp : cela permet donc d'utiliser ensuite n'importe quel solveur existant. La résolution d'un PLNE non-compacte est ici présentée au-travers du logiciel SCIP.

## 3.1 Le problème du sous-graphe acyclique induit

Soit un graphe orienté $G = (V, A)$ où $V$ est l'ensemble des sommets et $A$ l'ensemble des arcs. On note $uv$ un arc de $A$ allant du sommet $u$ au sommet $v$. On note *chemin* dans $G$ une suite d'arcs $P = (u_0u_1, u_1u_2, ..., u_{k-1}u_k)$, on dit alors que $P$ est de taille $k$. Un *circuit* $C$ dans $G$ est donc un chemin tel que $u_0 = u_k$. On note $V(P)$ (resp. $V(C)$) l'ensemble des sommets impliqué dans un chemin (resp. un circuit).
Un graphe est dit *acyclique* s'il ne contient aucun circuit. Soit $W \subset V$ un sous-ensemble de sommets, on note $A(W)$ l'ensemble des arcs ayant leurs deux extrémités dans $W$. On dit alors que le graphe $(W, A(W))$ est le graphe induit par $W$.
Le *problème du sous-graphe acyclique induit* (PSAI) consiste à déterminer un sous-graphe acyclique induit $(W, A(W))$ de $G$ tel que $|W|$ soit maximum. Ce problème a été montré comme étant NP-complet.

On considère les variables $x_i$ $\forall i \in V$ indiquant si le sommet $i$ est pris ou non dans la solution.

• *Formulation compacte :*

Soit un graphe orienté $G = (V, A)$, pour tout sommet $u \in V$. Considérons la formulation $(P_C)$ suivante :

$$
\begin{array}{llll}
Max & \displaystyle\sum_{u \in V} x(u) & & \\
 & u_1 = 1, & & (3.1) \\
 & 2 \leq u_i \leq n & \text{pour tout } i \neq 1, & (3.2) \\
 & u_i - u_j + 1 \leq n(2 - x_i - x_j) & \text{pour tout } i \neq 1, j \neq 1, & (3.3) \\
 & u_i \in I\!R & \text{pour tout} i, & (3.4) \\
 & 0 \leq x(u) \leq 1 & \forall u \in V & (3.5) \\
 & x(e) \text{ entier} & \forall u \in V. & (3.6)
\end{array}
$$

On peut montrer qu'elle est équivalente au PSAI. Elle est compacte mais possède une mauvaise relaxation linéaire qui rend cette formulation inefficace.

• *Formulation non-compacte :*

Soit un graphe orienté $G = (V, A)$, pour tout sommet $u \in V$. Considérons la formulation $(P_E)$ suivante :

$$
\begin{array}{llll}
Max & \displaystyle\sum_{u \in V} x(u) & & \\
 & \displaystyle\sum_{u \in V(C)} x(u) \leq |C| - 1, & \forall C \subset V \text{ tel que } (C, A(C)) & (3.7) \\
 & 0 \leq x(u) \leq 1, & \forall u \in V & (3.8) \\
 & x(e) \text{ entier}, & \forall u \in V. & (3.9)
\end{array}
$$

On appelle une telle inégalité (3.7) une contrainte de circuit. Cette formulation est équivalente au PSAI.

Cette formulation $(P_E)$ contient un nombre exponentiel de contraintes par rapport au nombre $n$ de sommets du graphe. Il n'est donc pas possible d'énumérer toutes ces contraintes pour ainsi créer un PLNE à entrer directement dans un solveur.
Par contre, il existe un algorithme de séparation polynomial pour les contraintes de circuit.

Pour cela, remarquons qu'en posant $x' = 1-x$, la contrainte devient alors $\sum_{u \in v(C)} x'(u) \geq 1$. Ainsi si l'on détermine un plus petit circuit $C^*$ selon le poids $x'$, on détecte soit : un cycle $C^*$ tel que $\sum_{u \in V(C^*)} x'(u) < 1$, c'est-à-dire une contrainte de circuit violée ; soit un cycle $C^*$ tel que $\sum_{u \in V(C^*)} x'(u) \geq 1$ et dans ce cas, cela signifie qu'il n'y a plus de contrainte de circuit violée par $x$.
La recherche d'un plus petit circuit se fait facilement en recherchant pour tout $u$ dans le graphe un plus court chemin selon $x'$ entre $u$ et un sommet $v$ voisin de $u$ qui ne passe par par l'arête $uv$.

## 3.2 Programmes et instances

Sur le site http ://www-desir.lip6.fr/∼fouilhoux/, section Documents d'enseignement et dans la section MADRO, vous trouverez l'archive Acyclic_Simple.tgz que vous pouvez décompresser. Cette archive est faite pour le système linux et un compilateur gcc. Mais elle peut être bien entendu adapter sous d'autres plateformes/compilateurs.

Une fois décompressée, l'archive propose une liste de fichiers et répertoires :
- createInstance : qui contient un générateur aléatoire d'instances pour le PSAI.
- graph : qui contient les fichier de manipulation d'une structure de données graphe très simple.
- compact_mip : qui contient les fichiers d'un programme créant un fichier .lp correspondant à la formulation compacte du PSAI, qui lance un solveur résolvant le PLNE du fichier puis qui lit le fichier .sol contenant la solution donnée par le programme.
- cutting_plane : qui contient les fichiser d'un programme SCIP permettant de résoudre la formulation non-compacte du PSAI
- bin : qui contient, après compilation, les exécutable du programme
- instance : répertoire vide où vous pouvez créer et stocker les instances et leurs solutions.

Les fichiers runme.sh et cleanme.sh permettent de lancer la compilation ou de détruire les programmes correspondant aux instances, graphes et PLNE compact (le cutting-plane est à part). Si ces fichiers ne sont pas exécutable après décompression : taper chmod +x runme.sh et chmod +x clean.sh.

Une fois compilé, vous avez dans le répertoire bin un exécutable createInstance qui permet de générer des instances du PSAI. Ce programme prend 3 paramètres : le nombre de sommets du graphe à générer, le degré maximum de ses sommets et un paramètre de génération (la graine de la générations aléatoire). Une instance est alors générées sous la forme d'un fichier txt de format assez simple (nombre de sommets,

puis pour chaque sommets la liste de ces successeurs qui se termine par -1).

## 3.3 Structures de données pour un graphe

Le répertoire graph contient les fichiers permettant de manipuler des fichiers :
- Graph.h et Graph.cpp : une classe graphe implémentant une structure de graphe par liste d'adjacence. Cette classe contient une implémentation simple mais efficace de l'algorithme de Dijkstra, ainsi que la recherche d'une plus petit circuit.
- dheap.h et dheap.cpp : une classe permettant de manipuler une structure de tas dédiée au code de Dijkstra.
- graphviewer.cpp : un code permettant de visualiser un graphe en écrivant sur disque un fichier pdf en utilisant le logiciel graphviz (voir documentation sur la même page web).
- solviewer.cpp : en lisant le fichier contenant un graph et celui contenant une solution issue du format .txt issu d'un solveur, ce programme permet d'afficher la solution.

## 3.4 PLNE compact

Le programme du répertoire compact_mip est très simple : il lit le fichier de graphe et crée un fichier au format .lp contenant la formulation compacte du PSAI correspondant.
Le programme lance ensuite un solveur : dans l'exemple il s'agit de SCIP, mais il est facile d'utiliser par exemple à la place glpk ou cplex ou ...
Le programme lit ensuite le fichier de sortie du solveur pour récupérer la solution et l'afficher.

• Le format .lp

Le format .lp est très simple. Il a été créé à l'origine pour le logiciel Cplex. Il revient à écrire un PL ou un PLNE sous le format habituel.Voici un petit exemple de programme

linéaire sous le format Cplex lp.

```
\Problem name: exemple.lp

Maximize
 obj: 5 x1 + 7 x2
Subject To
 c1: 3 x1 - 2 x2 <= -9
 c2: 10 x1 + 6 x2 >= 11
 c3: 8 x1 + x2  = -6
Bounds
 -Inf <= x1 <= 0
 1 <= x2 <= 4
End
```

Cet exemple se passe de commentaires, mais il faut noter que :
- la première ligne, optionnelle, indique un nom de problème
- les noms de la fonction obj ou des contraintes sont optionnels
- les variables peuvent prendre des noms quelconques
- Si une variable n'a pas borne inférieure dans la section Bounds, elle est considérée positive ou nulle.

Pour les PL mixtes ou entiers, on ajoute une section.
- Integers contenant la liste des variables entières
- Binary contenant la liste des variables binaires.

## 3.5 PLNE non compacte : Logiciel SCIP

Le répertoire cutting_plane contient une implémentation simple d'un algorithme de coupes et branchement pour la formulation non-compacte du PSAI.

### 3.5.1 Installation de SCIP

Le logiciel SCIP "Solving Constraint Integer Programs" est un framework pour la manipulation des algorithmes de coupes et génération de colonnes et branchement (branch-cut-and-price). Il est également un solveur entier performant et même le plus performant des solveurs non commerciaux. Il s'agit d'un produit universtaire de l'université ZIB de Berlin : http ://scip.zib.de/

SCIP fait en fait partie d'une suite logicielle la "SCIP Optimization Suite" que vous pouvez télécharger facilement sur la page de téléchargement.

Après téléchargement et décompression, il suffit de suivre les instructions contenues dans le fichier INSTALLATION. Une chose importante à retenir : la plupart des bugs d'installation proviennet du fait que SCIP utilise des librairies que vous n'avez peut-être pas.
Si certaines d'entre elles sont nécessaires, d'autres peuvent être non-utilisées, ce qui résoud l'essentiel des bugs : pour cela, utilisez la commande :
make ZIMPL=false ZLIB=false READLINE=false

pour chaque compilation.

Une fois compilé, vous pouvez accéder à l'exécutable scip dans le répertoire bin.
Vous pouvez également compiler et tester les exécutables correspondants à des exemples fournis avec SCIP : le TSP, coloration etc.

SCIP utilise en boîte noire un solveur de PL. Par défaut, il s'agit du solveur linéaire SoPlex de la ZIB, mais en suivant les instructions données, vous pouvez utiliser Cplex ou autres (les performances globales dépendent fortement de ce solveur).

### 3.5.2 Description d'un programme SCIP

Un programme SCIP nécessite un répertoir comme celui que vous avez dans cutting_plane :
- src : qui va contenir les fichers cpp et h source
- obj : qui va contenir les fichiers temporaires de compilation
- bin : qui va contenir les exécutables après compilation. Attention, dans cet exemple, ils sont en fait placé dans le répertoir bin un cran plus haut dans la hiérarchie.

Attention : Le Makefile doit être mis à jour pour correspondre à l'emplacement de SCIP sur votre disque.

Le répertoire src contient plusieurs fichiers essentiels :
- AcyclicMain.cpp : le programme principal qui contient la lecture du fichier, la création du PLNE, le lancement de la résolution, la récupération et affichage des résultats.
- AcyclicPLNE.cpp : une classe permettant de stocker tout ce qui est utile à l'exécution d'un PLNE : les données et les fonctions de création du PLNE.

### 3.5.3 La classe Constraint Handler

Les fichiers précédents sont être suffisants si votre formulation est compacte ! Dans le cas contraire, il faut aussi considér des classes Constraint Handler qui permettent d'implémenter un algorithme de séparation.

Dans cet exemple, nous avons codé la séparation de la contrainte de circuit dans la classe AcyclicCte.

La classe AcyclicCte contient plusieurs méthodes nécessaires :
- scip_check : qui renvoie vraie si la solution passée en paramètre est bien un sous-graphe acyclique. Attention, cette contrainte est nécessaire ici car un point entier peut-

être testée par SCIP dès qu'il en détecte un, que ce soit en allant très profondemment dans l'arbre de branchement, ou en le générant par une heuristique primale.
- scip_sepalp : qui recherche s'il existe des contraintes de cycles violées et qui les ajoutent dans ce cas. Cette méthode fonctionne ici en lançant deux fonctions spécifiques à notre problème : separ_AcylicCte qui lance find_AcyclicCte.
- scip_sepasol qui doit contenir le même code que scip_sepalp (elle est rarement utilisée par SCIP dans notre cas).
- scip_lock : cette fonction indique à SCIP ce qu'il est autorisé à faire pour fixer les valeurs des variables. En effet, SCIP tente de fixer des valeurs par inférences logiques, mais il ne peut le faire en théorie que s'il connaît toutes les contraintes de la formulation. Ce n'est pas le cas car la formulation n'est pas donnée au départ. On indique donc par cette contrainte le champ de possibilités pour chaque variable : Scip peut-il ou non augmenter/baisser la valeur d'une variable sans risquer de violer une contrainte de cette famille d'inégalités ? C'est le cas pour cette formulation : SCIP ne doit pas augmenter la valeur des variables. On indique cela au moyen de la fonction SCIPaddVarLocks() : si la contrainte peut être violée en baissant la valeur de la variable il faut appeler SCIPaddVarLocks(scip, var, nlockspos, nlocksneg) ; si la contrainte peut être violée en augmentant la valeur de la variable il faut appeler SCIPaddVarLocks(scip, var, nlocksneg, nlockspos) si la contrainte peut être violée en changeant la variable (+ ou -) il faut appeler SCIPaddVarLocks(scip, var, nlockspos + nlocksneg, nlockspos + nlocksneg).

### 3.5.4 Le Constraint Handler AcyclicCte

Le programme exemple Acyclic contient un seul constraint handler qui a pour but d'ajouter des contraintes de cycle. Pour cela, il contient - le code nécessaire à la recherche des contraintes de cycles dans find_AcyclicCte qui manipule le graphe.
- le code nécessaire à l'ajout des contraintes dans le programme dans separ_AcylicCte. Entre les deux fonctions, une liste de liste des variables concernées par les contraintes est passée en paramètre. Le fait de diviser cela en 2 fonctions permet de rendre le code plus simple à lire. Il est tout à fait possible de tout coder dans sepalp directement (attention, il faut le même code dans sepasol aussi).

### 3.5.5 Autres classes optionnelles

D'autes classes optionnelles permettent d'obtenir un code plus performant : implémentation d'un branchement particuliers, ajout de contraintes supplémentaires etc.

Dans cet exemple, une de ces classes est donnée en exemple :
AcyclicHeur qui correspond à une heuristique primale particulière pour le PSAI. Cette

heuristique est très simple : elle ajoute de manière gloutonne les sommets un à un en essayant de ne jamais former de circuits. Les sommets sont ajoutés dans l'ordre de leur poids selon la valeur $x$ de relaxation.

### 3.5.6 Les fonctions importantes

• Pour appeler une fonction SCIP, il est utile d'employer les macros SCIP_CALL, SCIP_CALL_ABORT et SCIP_CALL_EXC. Ces trois macros servent à appeler une fonction SCIP et s'utilisent de la même façon : la fonction SCIP est passée entre parenthèses comme argument de la macro. Par exemple : SCIP_CALL( uneFonctionScip() ) ;
Généralement, les fonctions SCIP retourne un code erreur du type SCIP_RETCODE qui vaut 1 si tout c'est bien passé, 0 si l'erreur n'est pas identifiée, un entier négatif pour une erreur spécifique (voir l'annexe). 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

• La fonction SCIPcreateVar permet de créer une variable. Voici le prototype de la fonction en laissant en non-affectée uniquement les paramètres les plus utilisés :
SCIP_RETCODE SCIPcreateVar(SCIP *scip, SCIP_CONS ** ptr_var, const char name, SCIP_Real lb, SCIP_Real ub, SCIP_Real coef, SCIP_VARTYPE vartype, TRUE, FALSE, NULL, NULL, NULL, NULL, NULL)) ;
- scip est un pointeur sur l'environnement scip qui est créé par la fonction SCIPcreate
- ptr_var est retourné par la fonction en contenant l'adresse de la variable. Cette variable peut ensuite être utilisée dans la fonction SCIPaddVar pour ajouter la variable au programme. Il est utile de conserver ce pointeur pour la suite (par exemple dans un tableau).
- lb et ub sont les bornes inf et sup de la variable
- coef est le coefficient de la variable dans la fonction objective
vartype donne le type de la variable entre SCIP_VARTYPE_BINARY, SCIP_VARTYPE_INTEGER, SCIP_VARTYPE_IMPLINT (variable continue mais qui est implicitement toujours entière) et SCIP_VARTYPE_CONTINUOUS.

• La fonction SCIPcreateConsLinear permet de créer une contrainte linéaire. Voici le prototype de la fonction en laissant en non-affectée uniquement les paramètres utilisés dans le cas d'un branch-and-cut (il faut prendre en compte les autres dans des cadres

comme par exemple le Branch-and-Price)
SCIP_RETCODE SCIPcreateConsLinear (SCIP *scip, SCIP_CONS ** ptr._cons, const char name, 0, NULL, NULL, SCIP_Real lhs, SCIP_Real rhs, TRUE, SCIP_Bool separate, FALSE, TRUE, SCIP_Bool propagate, SCIP_Bool local, FALSE, SCIP_Bool dynamic, SCIP_Bool removable, FALSE) ; - ptr_cons est un pointeur sur la contrainte. Cette variable peut être ajoutée au programme avec SCIPaddCons. Il est rarement utile de conserver ce pointeur, il vaut mieux alors le libérer immédiatement après avec SCIPreleaseCons.
- La contrainte est en fait de la forme $lhs \leq ax \leq rhds$, si l'on désire n'avoir que le lhs ou le rhs, on fixe alors une des bornes à la valeur -SCIPinfinity(scip) ou +SCIPinfinity(scip).
- separate TRUE si la contrainte sera séparée par la suite
- propagate TRUE si l'on désire que les contraintes soient utilisées dans les problèmes-fils du noeud de branchement où la contrainte a été créée.
- local TRUE si la contrainte n'est vraie que dans le noeud de branchement (et éventuellement dans ses fils. FALSE pour que la contrainte soit utilisée dans tout le programme.
- dynamic TRUE si cette contrainte peut se retrouver un jour concernée par le cas décrit au paramètre removable
- removable TRUE si l'on autorise SCIP à supprimer cette contrainte du programme au bout d'un certain temps où elle n'a pas servi à former un point optimal

• La contrainte créée par la fonction précédent ne contient pas de variables. On les ajoute en utilisant SCIPaddCoefLinear qui utilise un pointeur sur la variable et un pointeur sur la contrainte.

# Chapitre 4

# Solutions approchées et garantie

Dans le but d'obtenir des algorithmes de branchement performants, il est nécessaire de rechercher de bonnes solutions pour le problème. Evidemment, cette démarche est également utile d'une manière générale.

## 4.1 Trouver des solutions réalisables

On dit souvent que tous les moyens sont bons pour découvrir une bonne solution d'un problème complexe comme les PMD. On peut diviser ces "moyens" en plusieurs catégories :
• heuristique spécifiques : suivant les problèmes une idée heuristique conduit parfois à de bonnes solutions (par exemple reproduire l'expérience d'un spécialiste)
• heuristique gloutonne : c'est le cadre des heuristiques où l'on ne remet jamais en question une décision prise précédemment. En général, pour les problèmes difficiles (NP-difficiles), ces heuristiques sont peu performantes. Il faut néanmoins noter qu'elles sont souvent les seules que l'on peut utiliser dans les cas où l'on doit traiter des données en temps réel (cas online). D'autre part, ces heuristiques sont parfois optimales (par exemple l'algorithme de Prim pour la recherche d'arbre couvrant).
• méta-heuristiques : il s'agit du cadre d'heuristiques qui reproduisent des principes généraux d'amélioration de solutions réalisables. Elles sont souvent utilisables pour des instances de très grandes tailles (là où les méthodes exactes échouent). On les divise en général entre :

- méthodes itératives qui tentent de reproduire le mécanisme des méthodes d'optimisation continue. Elles reposent sur la définition de voisinage : c'est-à-dire de déterminer comment passer d'une solution réalisable à une autre solution réalisable "voisine" dans le but d'en déterminer une meilleure. Les plus célèbres sont la descente stochastique, le recuit simulé et la méthode Taboo.
- méthodes évolutives Elles reproduisent des phénomènes biologiques ou physiques

où l'on voit apparaître des solutions noouvelles en "mélangeant" des informations provenant d'un lot de solutions réalisables. A chaque itération de cette méthode, on obtient ainsi un nouveau lot de solutions réalisables construits à partir du lot précédent. Les plus célèbres : algorithme génétique, les colonies de fourmis, les essaims de particules...

• heuristique primale : dans le cadre des PMD, une relaxation continue fournie une solution fractionnaire (que l'on appelle solution primale) qui peut-être "proche" d'une solution entière réalisable. On peut ainsi imaginer en quelque sorte d'arrondir ces solutions fractionnaires. (Par exemple, dans le cas du problème de sac-à-dos : l'arrondi de la solution franctionnaire à l'entier inférieur donne une solution réalisable.) Ces méthodes s'appellent ainsi parfois *méthode d'arrondi* (rouding). Suivant le problème et l'arrondi utilisé, cette méthode ne fournit pas toujours une solution réalisable : il faut alors soit la corriger, soit la refuser. Il est aussi possible de "tirer au sort" un arrondi (random rounding).

On peut noter également que les moteurs d'inférences de la programmation par contrainte permettent parfois de déduire des solutions réalisables de manière générique et efficace. Dans ces cas, on peut construire des algorithmes de branchement efficace (appelés parfois méthode Dive&Fix).

## 4.2 Solution à garantie théorique

Les méthodes précédentes ont été décrites dans le but de donner des idées pour produire des solutions réalisables dites "approchées". Il est bien plus intéressant de produire des solutions pour lesquelles on peut indiquer son éloignement à l'optimum. On parle alors de *solutions garanties* (provably good solutions).

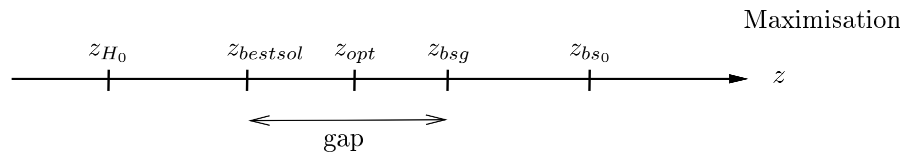
• *Relation Min-Max* : on appelle une relation Min-Max un encadrement de la solution optimale d'un problème par une borne min et une borne max obtenue par un même algorithme. Cet algorithme est le plus souvent obtenu en utilisant la théorie de la dualité ou des propriétés combinatoires particulière.

• *Algorithme d'approximation* : on appelle algorithme d'$\alpha$-approximation un algorithme (polynomial ou à la rigueur efficace) donnant une solution qui est au pire à $\alpha\%$ de l'optimum. Dans le cas d'une maximisation, cela siginifie que la valeur $z_{approx}$ est telle que $\alpha z_{opt} \leq z_{approx} \leq z_{opt}$.
Ces algorithmes sont le plus souvent très spécifiques et sont basés sur des principes algorithmes parfois très fins. Il est également possible que des technique d'arrondis ou des heuristiques primales puissent fournir des algorithmes d'approximation intéressant.

Ces techniques demandent des preuves théoriques poussées et fournissent ainsi des algorithmes efficaces à garantie théorique. En revanche, ils ne donnent que rarement des solutions intéressantes.

## 4.3 Solution approchée avec garantie expérimentale

On parle de *garantie expérimentale* dans le cas où un algorithme permet de donner à la fois une solution approchée et une borne sur son écart à l'optimum. C'est le cas pour les algorithmes de branchement-évaluation. En effet, dans le cas maximisation, si l'on dispose d'une solution réalisable (borne inférieure) produite par une méthode heuristique initiale $z_{H_0}$ et d'une évaluation du problème $z_{bs_0}$, on obtient un encadrement de la solution optimale $z_{opt}$ : $z_{H_0} \leq z_{opt} \leq z_{bs_0}$. Au cours de l'algorithme, on découvre de nouvelles solutions réalisables et de meilleure borne (supérieure). En fait, à tout moment, on peut considérer la meilleure solution réalisable rencontrée $z_{bestsol}$ et la plus petite valeur de borne supérieure globale rencontrée $z_{bsg}$. On peut donc considérer tout au long du déroulement de l'algorithme l'augmentation de la borne inférieure et la réduction de la borne supérieure : cela reduit l'écart.




On mesure cet écart en tant qu'écart relatif appelé souvent *gap* :

$$gap = \frac{z_{bsg} - z_{bestsol}}{z_{bestsol}}.$$

Ainsi, dans le pire des cas, la borne supérieure est égale à $z_{opt}$, ce qui signifie que la meilleure solution réalisable trouvée est au plus loin de l'optimum : on aurait ainsi $(1 + gap)z_{bestsol} = z_{opt}$ ce qui signifie qu'au pire $z_{bestsol}$ est à $gap\%$ de l'optimum.

On peut ainsi obtenir à tout moment une garantie expérimentale sur les solution approchées. Si le gap est peut important, par exemple inférieur à 5%, on peut souvent considérer que l'on a obtenu une très bonne solution (éventuellement même optimale !) : industriellement, les valeurs que l'on manipulent en entrée du problème sont parfois imprécises ou soumises à de fortes variations, il n'est pas systématiquement nécessaire d'obtenir une meilleure solution ! Inversement, si le gap est mauvais (supérieur à 5%), cela peut-être dû au fait que la solution approchée trouvée est mauvaise... mais cela peut aussi provenir d'une mauvaise évaluation.