UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ÉTUDE DE LA SÉPARATION DES INÉGALITÉS VALIDES POUR LE PROBLÈME DE TOURNÉES DE VÉHICULES AVEC PLUSIEURS DÉPÔTS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

PAR

MOUNIRA GROIEZ

JUIN 2007

REMERCIEMENTS

Mes vifs remerciements à M^{me} Odile Marcotte pour son encadrement, sa bienveillance, sa disponibilité, sa confiance, ses précieux conseils et ainsi pour la bourse qu'elle m'a attribuée.

Je remercie M. Ahmed Hadjar, pour toute l'aide qu'il m'a apportée, pour sa gentillesse et pour sa disponibilté.

Je remercie les membres du jury qui m'ont fait l'honneur d'accepter de juger mon travail.

Je remercie tout le personnel du GERAD, en particulier Pierre, Serge et Francine pour leur aide.

Je remercie Manon Gauthier pour toute l'aide qu'elle m'a apportée durant mes années d'études à l'UQAM.

Je remercie les membres de ma famille et mes amis, en particulier mon frère Fayçal, Sarah, Khadidja, Amira, Alain et Mbaye, pour leur patience, leur gentillesse et pour leurs encouragements.

Mes très chaleureux remerciements à mes parents, qui tout au long de ma vie, ont su m'encourager et trouver les mots et les moyens pour m'aider et me soutenir.

Qu'ils trouvent tous ici, le témoignage de ma reconnaissance et l'expression de ma profonde gratitude.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Le problème de tournées de véhicules avec plusieurs dépôts et des dates fixes pour les trajets (nommé MDVSP) est un des problèmes les plus importants en optimisation combinatoire ; le MDVSP est un problème NP-complet.

Dans le présent mémoire nous présentons un algorithme de type « Branch-and-Cut » pour la résolution du MDVSP. Notre algorithme combine la méthode de séparation et évaluation progressive et une procédure de séparation des inégalités valides, qui consiste à détecter certains trous impairs et à les lifter. Nous présentons aussi les résultats des tests effectués et l'analyse des comparaisons entre notre algorithme et les méthodes utilisées par le logiciel CPLEX.

Mots-clefs : Le problème de tournées de véhicules avec plusieurs dépôts, séparation des inégalités valides, trous impairs, algorithme de « Branch-and-Cut » pour la résolution du MDVSP

INTRODUCTION

Le problème de tournées de véhicules avec plusieurs dépôts et des heures fixes pour les trajets (dit MDVSP) est parmi les problèmes d'optimisation combinatoire les plus importants sur les plans pratique et théorique.

En pratique, généralement le nombre de trajets à effectuer est très important et les coûts associés sont très élevés, ce qui pousse les compagnies de transport à chercher la meilleure manière possible de minimiser les coûts tout en respectant leurs engagements envers les clients, ainsi que leurs contraintes de travail.

En théorie et grâce aux techniques de la recherche opérationnelle, le MDVSP est formulé sous forme de programme linéaire en nombres entiers (la formulation dépend du problème pratique étudié, mais les variables sont toujours entières). Les programmes linéaires généraux en nombres entiers sont NP-complets, donc il n'existe pas encore d'algorithme simple permettant la résolution de tels problèmes en temps polynomial. Le MDVSP est lui-même NP-complet. Pour le résoudre, il existe des algorithmes (ou méthodes) qui combinent la séparation et évaluation progressive, la génération de colonnes, la génération de contraintes et, dans certains cas, d'autres procédures afin d'accélérer le processus de résolution. Il existe aussi plusieurs heuristiques pour résoudre le MDVSP.

Notre objectif était de migrer une partie du travail effectué par Hadjar, Marcotte et Soumis (2006) de GENCOL (un optimiseur commercial de génération de colonnes) à CPLEX (un optimiseur commercial de programmation linéaire en nombres entiers), c'est-à-dire d'implémenter leur algorithme sans utiliser la génération de colonnes, tout en essayant de détecter les trous impairs les plus importants. La notion de « trou impair » sera définie dans le chapitre III. Nous présentons dans ce travail un algorithme de type « Branch-and-Cut » pour la résolution du MDVSP. Notre algorithme combine

la méthode de séparation et évaluation progressive et une procédure de séparation des inégalités valides, qui consiste à détecter certains trous impairs et à les « lifter » (enrichir).

Dans le présent document, nous commençons par un bref survol de la littérature dans le premier chapitre. Le deuxième chapitre contient quelques notions nécessaires et des méthodes de résolution des programmes linéaires en nombres entiers. La méthode de détermination des inégalités valides pour le MDVSP fait l'objet du troisième chapitre. Dans le dernier chapitre, nous présentons les résultats des tests que nous avons effectués et nous terminons par une conclusion.

CHAPITRE I

SURVOL DE LA LITTÉRATURE

1.1 Introduction

Le MDVSP a fait l'objet de plusieurs travaux de recherche durant les dernières années. Plusieurs algorithmes ont été proposés utilisant la séparation et évaluation progressive, le « Branch-and-Cut », la génération de colonnes, la relaxation lagrangienne, etc.

Bertossi, Carraresi et Gallo (1987) ont montré que le MDVSP est NP-complet. Mais dans le cas où le nombre de dépôts est égal à 1, le problème peut être résolu en temps polynomial. Carpaneto, Dell'Amico, Fischetti et Toth (1989) ont observé que le MDVSP peut être aussi résolu en temps polynomial si les coûts de transfert entre les dépôts et les tâches sont indépendants des dépôts. Plusieurs algorithmes ont été proposés pour la résolution du MDVSP ; nous trouverons dans l'article de Desaulniers et Hickman (2007) une revue de littérature regroupant les plus récents et plus importants travaux publiés sur le problème. Dans le présent chapitre, nous nous contenterons d'en présenter quelques-uns.

Considérons un ensemble de n trajets (ou tâches) $\{T_1, T_2,...,T_n\}$, où chaque trajet T_i $(i = 1, 2, \ldots, n)$ débute au temps a_i et se termine au temps b_i. Les véhicules effectuant les trajets sont logés dans des dépôts. Soit K l'ensemble des indices des dépôts considérés, où $K = \{1, 2, ..., |K|\}$. Nous supposons que le dépôt D_k a une capacité de v_k véhicules (pour tout k). Soit t_{ij} le temps nécessaire à un véhicule pour se rendre de l'emplacement

où le trajet T_i se termine à l'emplacement où le trajet T_j doit commencer. La paire de trajets consécutifs (T_i, T_j) est dite compatible (ou réalisable) si un véhicule peut effectuer T_j immédiatement après T_i, c'est-à-dire que la relation $b_i + t_{ij} \leq a_j$ est vérifiée.

Le coût associé à chaque paire de trajets compatible (T_i, T_j) est $c_{ij} \geq 0$; notons que $c_{ij} = +\infty$ si (T_i, T_j) est non réalisable ou $i = j$. Pour chaque trajet T_i et chaque dépôt D_k, soit $c_{n+k,i}$ (resp. $c_{i,n+k}$) le coût non négatif engendré par l'utilisation d'un véhicule du dépôt D_k débutant (resp. terminant) sa tournée par le trajet T_i. Par conséquent, le coût total d'un chemin $(T_{i_1}, T_{i_2},..., T_{i_h})$ associé à un véhicule du dépot D_k est $c_{n+k,i_1} + c_{i_1,i_2} + \ldots + c_{i_h,n+k}$.

Le MDVSP consiste à trouver une affectation réalisable des trajets de coût minimal possédant les propriétés suivantes :

- chaque trajet est effectué par exactement un véhicule ;
- chaque véhicule utilisé dans la solution couvre une suite de trajets (un chemin) dans laquelle toute paire de trajets consécutifs est compatible (réalisable) ;
- chaque véhicule, après avoir effectué ses tâches, revient au dépôt de départ ;
- le nombre de véhicules du dépôt D_k utilisés ne peut pas dépasser v_k (la capacité du dépôt D_k) ;
- le coût total de l'affectation est égal à la somme des coûts des chemins effectués par les véhicules.

1.2 Algorithme de séparation et évaluation pour le MDVSP (Carpaneto, Dell'Amico, Fischetti et Toth (1989))

Le problème du MDVSP a été formulé par ces auteurs comme suit. Considérons un graphe $G = (V, A)$ tel que $V = \{1, 2, ..., p\}$ (où $p = n + \sum_{k=1}^{|K|} v_k$) est l'ensemble des sommets, partitionné en $|K| + 1$ sous-ensembles : le sous-ensemble $N = \{1, ..., n\}$, dans lequel le sommet j est associé au trajet T_j, et les sous-ensembles $W_1, W_2, ..., W_{|K|}$, où pour $k = 1, ..., |K|$, $W_k = \left\{ n + \sum_{h=1}^{k-1} v_h + 1, ..., n + \sum_{h=1}^{k} v_h \right\}$ contient v_k sommets associés

aux v_k véhicules du dépôt D_k. $A = \{(i,j) \mid i,j \in V\}$ dénote l'ensemble des arcs. Le coût associé à chaque arc appartenant au graphe $G = (V, A)$ est défini comme suit :

$$c'_{ij} = \begin{cases} c_{ij} & \text{si } i,j \in N \\ c_{n+k,j} & \text{si } i \in W_k, j \in N \\ c_{i,n+k} & \text{si } i \in N, j \in W_k \\ 0 & \text{si } i,j \notin N, i = j \\ \infty & \text{si } i,j \notin N, i \neq j \end{cases}$$

Afin de présenter la formulation mathématique du problème, nous avons besoin de définir la variable binaire x_{ij} qui prend la valeur 1 si l'arc $(i,j) \in A$ est utilisé dans la solution optimale et la valeur 0 sinon. Soit Π la famille de tous les chemins élémentaire $\mathcal{P}$ reliant les différents sommets dans $V \setminus N$. Voici la formulation, dont la valeur optimale est dénotée $v(MDVSP)$.

$$(MDVSP) \begin{cases} \text{minimiser} & \displaystyle\sum_{i \in V} \sum_{j \in V} c'_{ij} x_{ij} & & (1) \\ \text{s.c.} & \displaystyle\sum_{i \in V} x_{ij} = 1, & j \in V & (2) \\ & \displaystyle\sum_{j \in V} x_{ij} = 1, & i \in V & (3) \\ & \displaystyle\sum_{(i,j) \in \mathcal{P}} x_{ij} \leq |\mathcal{P}| - 1, & \mathcal{P} \in \Pi & (4) \\ & x_{ij} \in \{0,1\}, & i,j \in V & (5) \end{cases}$$

Les contraintes (2) et (3) imposent que chaque sommet $k \in V$ est visité une seule fois. La contrainte (4) permet la suppression des chemins non réalisables, c'est-à-dire les chemins qui comprennent plus d'un dépôt (notons que $|\mathcal{P}|$ représente le nombre d'arcs de $\mathcal{P}$).

Carpaneto, Dell'Amico, Fischetti et Toth (1989) ont proposé un algorithme de type séparation et évaluation progressive basé sur la borne inférieure additive Z_{ADD}, qui est calculée comme suit. Initialement, la valeur de Z_{ADD} est égale à la valeur optimale

du problème d'affectation (AP), obtenue en supprimant la contrainte (4) du problème ($MDVSP$). Ensuite, pour chaque dépôt D_k et pour chaque tâche T_j, nous calculons $\delta_{kj} = \lambda_{kj} + \bar{\lambda}_{kj}$, où λ_{kj} (resp. $\bar{\lambda}_{kj}$) est la longueur du plus court chemin du dépôt D_k (resp. le sommet associé à la tâche T_j) au sommet associé à la tâche T_j (resp. dépôt D_k) ne contenant aucun autre dépôt. La nouvelle valeur de Z_{ADD} est $Z_{ADD} + \delta_{p\ell}$, où $\delta_{p\ell} = \text{maximum}_{k=1,...|K|;j=1,...,n}\{\delta_{kj}\}$. La solution duale du problème du plus court chemin du dépôt D_p à la tâche T_ℓ permet de définir les nouvelles valeurs des coûts réduits du primal. L'itération suivante consiste à calculer la nouvelle borne du plus court chemin en terme de coûts réduits, jusqu'à ce qu'aucune amélioration ne puisse être effectuée.

1.3 Approche de génération de colonnes (Soumis et Ribeiro (1994))

Soumis et Ribeiro (1994) ont présenté une nouvelle formulation mathématique du MDVSP, et ils ont montré que la relaxation linéaire de cette formulation fournit une meilleure borne inférieure que celle obtenue par la technique de bornes additives utilisée par Carpaneto *et al.* (1989). Ils ont aussi présenté une nouvelle formulation du problème et utilisé la technique de génération de colonnes pour le résoudre.

La formulation mathématique du MDVSP utilisée par les auteurs est celle d'un modèle polyvalent de flot dans les réseaux. Soit $N = \{1, ..., n\}$ l'ensemble des tâches et $K = \{1, ..., |K|\}$ l'ensemble des dépôts. À chaque dépôt $k \in K$, nous associons un graphe $G^k = (V^k, A^k)$, où $n + k$ indique le dépôt k, $V^k = N \cup \{n + k\}$ et $A^k = (N \times N) \cup (\{n + k\} \times N) \cup (N \times \{n + k\})$. Soit x_{ij}^k le flux de type k à travers l'arc $(i, j) \in A^k$. Le problème du MDVSP se formule comme suit :

$$
(MDVSP)\begin{cases}
\text{minimiser} & \displaystyle\sum_{k=1}^{|K|}\sum_{(i,j)\in A^k} c_{ij}x_{ij}^k & (1)\\
\text{s.c.} & \displaystyle\sum_{k=1}^{|K|}\sum_{i\in V^k} x_{ij}^k = 1, \quad j\in N & (2)\\
& \displaystyle\sum_{i\in V^k} x_{ij}^k - \sum_{i\in V^k} x_{ji}^k = 0, \quad \forall k\in K, \forall j\in V^k & (3)\\
& \displaystyle\sum_{j\in N} x_{n+k,j}^k \le v_k, \quad \forall k\in K & (4)\\
& x_{ij}^k \ge 0, \quad \forall k\in K, \forall (i,j)\in A^k & (5)\\
& x_{ij}^k \text{ entier} \quad \forall k\in K, \forall (i,j)\in A^k. & (6)
\end{cases}
$$

Soumis et Ribeiro (1994) ont reformulé le problème du MDVSP comme un problème de partitionnement d'ensemble. Pour chaque $k \in K$, soit Ω_k l'ensemble des chemins débutant au dépôt D_k, visitant certains sommets de N et retournant au même dépôt. Pour chaque chemin $p \in \Omega = \cup_{k=1}^{|K|}\Omega_k$, soit c_p la somme des coûts des arcs du chemin p et a_{jp} une constante binaire prenant la valeur 1 si et seulement si le sommet $j \in N$ appartient à p. En associant la variable binaire y_p à chaque chemin $p \in \Omega$, le problème se formule comme suit :

$$
(MDVSP')\begin{cases}
\text{minimiser} & \displaystyle\sum_{k=1}^{|K|}\sum_{p\in\Omega_k} c_p y_p & (7)\\
\text{s.c.} & \displaystyle\sum_{k=1}^{|K|}\sum_{p\in\Omega_k} a_{jp}y_p = 1, \quad j\in N & (8)\\
& \displaystyle\sum_{p\in\Omega_k} y_p \le v_k, \quad \forall k\in K & (9)\\
& y_p \in \{0,1\}, \quad \forall p\in\Omega. & (10)
\end{cases}
$$

La relaxation continue de $(MDVSP')$ est résolue par la génération de colonnes. Cette dernière est utilisée pour résoudre un problème linéaire qui contient un nombre très important de variables. La technique de génération de colonnes est une méthode itérative qui résout alternativement deux problèmes : en premier, un problème maître

restreint, qui est une version restreinte du problème original avec un très petit nombre de variables ; ensuite un sous-problème, en général facile à résoudre, qui permet l'identification des variables à ajouter au problème maître restreint qui sera résolu à nouveau. La méthode converge en un nombre fini d'itérations et s'arrête lorsque le sous-problème n'identifie aucune variable à ajouter au problème maître.

Pour le MDVSP, la résolution d'un sous-problème consiste à trouver un plus court chemin partant du dépôt D_k, visitant un certain ensemble de sommets et retournant au même dépôt. Si les coûts réduits des plus courts chemins de chaque dépôt D_k sont tous non négatifs, alors la solution courante est optimale. Par contre, s'il existe au moins un plus court chemin de coût réduit négatif, la variable correspondante est ajoutée au problème qui est résolu de nouveau.

1.4 L'algorithme de type « Branch-and-Cut » pour la résolution du MDVSP (Hadjar, Marcotte et Soumis (2006))

Hadjar, Marcotte et Soumis (2006) ont proposé un algorithme de type « Branch-and-Cut », combinant une procédure de type séparation et évaluation progressive, une procédure permettant la fixation de certaines variables, la génération de colonnes et la génération de contraintes. Ils ont aussi montré que la relaxation linéaire du MDVSP contient beaucoup de cycles impairs. Dans notre travail, nous avons adopté la formulation utilisée par ces auteurs, identique à celle de Soumis et Ribeiro (1994).

En premier, une heuristique est appliquée afin de déterminer une solution entière heuristique. Elle consiste à calculer la solution optimale X de la relaxation linéaire au noeud racine. À partir de cette solution un groupe de paires (T_i, T_j), noté A_p, est choisi de telle sorte que les relations $|A_p| \geq 1$ et $\sum_{(T_i,T_j)\in A_p} \left(1 - \sum_{k=1}^{|K|} X_{ij}^k\right) < 0.75$ soient vérifiées. Pour fixer les paires (T_i, T_j) appartenant à A_p (c'est-à-dire appliquer la contrainte $\sum_{k=1}^{|K|} X_{ij}^k = 1$), l'algorithme supprime tous les arcs sortant de T_i et aboutissant à T_j, sauf ceux de la forme (T_i, T_j, k), dans chaque sous-problème. Ainsi un noeud fils du noeud

racine est créé en ajoutant la contrainte $\sum_{k=1}^{|K|} X_{ij}^k = 1$ pour chaque paire $(T_i, T_j) \in A_p$ et l'algorithme résout la nouvelle relaxation linéaire. Si l'algorithme trouve des plans coupants, un noeud fils du nouveau noeud est créé et les plans coupants sont ajoutés au modèle. Par contre, s'il n'existe pas de plans coupants, l'algorithme crée un noeud fils du nouveau noeud en fixant certaines paires (T_i, T_j) de la même manière qu'au noeud racine. L'heuristique procède de cette façon jusqu'à l'obtention d'une solution entière.

La solution entière heuristique est exploitée pour fixer des variables du programme linéaire au noeud racine, et ces variables sont supprimées définitivement du problème avant que la méthode de génération de colonnes ne soit appliquée. La méthode de fixation des variables est utilisée par la suite au niveau de chaque noeud de l'arbre de branchement en utilisant la meilleure solution entière trouvée. La stratégie utilisée pour fixer les variables a permet de supprimer plus de 90% de variables de la plupart des instances traitées par Hadjar et *al.* (2006).

Au niveau de chaque noeud de l'arbre de branchement, l'algorithme résout la relaxation linéaire par la méthode de génération de colonnes. Dans le but de diminuer le nombre de variables, la méthode de fixation des variables est appliquée. Ensuite, l'algorithme détermine les plans coupants correspondants aux cycles impairs du MDVSP. Les inégalités valides sont « liftées » à l'aide d'un algorithme glouton, proposé par les auteurs, avant de les ajouter au problème. Hadjar, Marcotte et Soumis (2006) ont montré que sous certaines conditions les inégalités valides liftées définissent des facettes de l'enveloppe convexe de l'ensemble des solutions du MDVSP. Pour le branchement, une heuristisque a été proposée par les auteurs de l'article et ils ont montré que dans certains cas, leur heuristique est meilleure que d'autres méthodes existantes. L'algorithme utilise plusieurs stratégies de branchement. Les auteurs ont réussi à résoudre des instances générées aléatoirement de 750 tâches et avec 6 dépôts, ainsi que d'autres instances en utilisant des données réelles. Les résultsts de leurs expériences montrent que leur algorithme est meilleur que CPLEX pour la plupart des instances testées.

1.5 Autres études

Il existe plusieurs autres articles qui ont étudié le problème du MDVSP. Un algorithme de type « Branch-and-Bound » a été décrit par Carpaneto et *al.*(1989). Fischetti, Dell'Amico et Toth (1993) ont proposé une heuristique. Forbes, Holt et Watts (1994) ont analysé la formulation en programme linéaire à trois indices. Löbel (1997) a étudié la décomposition de Dantzig-Wolfe appliquée à la formulation avec trois indices. Fischetti, Lodi, Martello et Toth (2001) ont présenté une étude polyédrale pour simplifier le problème du MDVSP. Pepin, Desaulniers, Hertz et Huisman (2006) ont comparé les performances d'une heuristique de programmation en nombres entiers, d'une heuristique lagrangienne, d'une heuristique de génération de colonnes, d'une méthode de recherche à grands voisinages utilisant la génération de colonnes et d'une méthode de recherche tabou.

Et il existe aussi d'autres algorithmes exacts et heuristiques qui ont fait l'objet de plusieurs articles.

CHAPITRE II

LA MÉTHODE DU « BRANCH-AND-CUT »

2.1 Introduction

Notre méthode de résolution du MDVSP, qui sera présentée dans le chapitre suivant, est basée principalement sur la méthode de « Branch-and-Cut ». Nous avons d'abord besoin de présenter la méthode de séparation et évaluation progressive, ensuite la méthode des plans coupants, avant de parler de la méthode de « Branch-and-Cut ». Nous terminerons par un exemple d'application de cette méthode.

2.2 Quelques notions de programmation linéaire en nombres entiers

Soit $x = (x_1, x_2, ..., x_n)$ et $y = (y_1, y_2, ..., y_n)$ deux vecteurs (de même dimension). Dans ce qui suit, la notion « $x \geq y$ » est équivalente à « $x_i \geq y_i$ » pour tout $i = 1, 2, ..., n$

Définition 2.2.1

Soit A une matrice de dimension $m \times n$, x un vecteur colonne à m variables, b un vecteur colonne de taille m et c un vecteur ligne de taille n, que nous supposons tous à composantes entières. Le problème

$$(P) \begin{cases} \min \quad z = cx \\ \quad \text{s.c.} \quad Ax \geq b \\ \qquad\quad x \in \mathbb{Z}^n, \ x \geq 0 \end{cases}$$

est appelé programme linéaire en nombre entiers.

Définition 2.2.2

Le problème

$$(P') \begin{cases} \min \quad z = cx \\ \quad \text{s.c.} \quad\quad Ax \geq b \\ \qquad\qquad\quad x \geq 0 \end{cases}$$

est appelé la relaxation linéaire de (P). Il est obtenu en relaxant les contraintes d'intégrité sur les variables.

Définition 2.2.3

L'ensemble $S = \{Ax \geq b \mid x \geq 0 \ \ (ou \ \ x \in \mathbb{Z}_+^n)\}$ est dit ensemble des solutions réalisables.

Définition 2.2.4

$conv(S)$ dénote l'enveloppe convexe de l'ensemble $S \subseteq \mathbb{R}^n$, c'est-à-dire le plus petit ensemble convexe de $\mathbb{R}^n$ qui contient S (où S est l'ensemble des solutions réalisables).

Définition 2.2.5

1. *$x = (x_1, x_2, ..., x_n)^T$ est une solution réalisable de (P') (resp. (P)) si et seulement si $Ax \geq b$ et $x \geq 0$ (resp. $x \in \mathbb{Z}_+^n$).*
2. *Une solution optimale de (P') (resp. (P)) est une solution réalisable qui donne à la fonction $Z = cx$ la plus petite valeur possible.*

Définition 2.2.6

Etant donné un programme linéaire en nombres entiers (P), on dit que l'inéquation $dx \geq a$ est valide si elle est satisfaite par tout point de S, où S est l'ensemble des solutions réalisables de (P).

Définition 2.2.7

Une coupe (ou plan coupant) est une inéquation valide qui n'est pas satisfaite par tout point de S', où S' est le domaine des solutions réalisables de la relaxation (P') de (P).

Définition 2.2.8

Un polyèdre P est un ensemble de points défini par une collection finie d'équations et d'inéquations linéaires ; il s'écrit sous la forme

$$P = \left\{ x \in \mathbf{R}^n \mid Ax \left\{ \begin{array}{c} \geq \\ = \end{array} \right\} b \right\},$$

où $A \in \mathbf{R}^{m \times n}$ et $b \in \mathbf{R}^m$.

Définition 2.2.9

Un polytope est un polyèdre borné.

Définition 2.2.10

Si l'intersection entre P et $\{x \in \mathbf{R}^n \mid dx = a\}$ est non vide et différente de P, et P est un sous-ensemble de $\{x \in \mathbf{R}^n \mid dx \geq a\}$, alors $F = \{x \in P \mid dx = a\}$ est dite face de P définie par l'inéquation valide $dx \geq a$.

Définition 2.2.11

Les faces F de dimension maximale, c'est-à-dire telles que $dim(F) = dim(P) - 1$, sont appelées facettes de P.

2.3 La méthode de séparation et évaluation progressive (Branch-and-Bound)

Le premier algorithme de séparation et évaluation progressive a été proposé en 1960 par Land et Doig, pour résoudre les programmes linéaires en nombres entiers. Cependant, la popularité de cette approche a augmenté après la publication de *« An algorithm for the traveling salesman problem »* par Little *et al.* (1963), parce qu'il a été démontré que beaucoup de problèmes peuvent être résolus par énumération implicite. Et en 1965, Balas a proposé le premier algorithme d'énumération implicite pour les programmes linéaires en variables binaires (c'est-à-dire où les variables appartiennent à l'ensemble $\{0, 1\}$).

La méthode de séparation et évaluation progressive est utilisée pour résoudre les programmes linéaires en nombres entiers. Le principe de la méthode est basé sur deux concepts : **le branchement**, qui consiste à diviser un ensemble de solutions en plusiseurs sous-ensembles, et **l'évaluation**, qui consiste à borner les valeurs des solutions appartenant à un sous-ensemble.

La méthode commence par résoudre la relaxation linéaire du problème. La valeur de la fonction-objectif obtenue est toujours inférieure ou égale à la valeur de la solution optimale du programme linéaire en nombres entiers, puisque toute solution de ce dernier est une solution de la relaxation linéaire. Par conséquent, si la solution optimale de la relaxation (dénotée x^0) est entière, alors c'est une solution optimale du programme linéaire en nombres entiers. Sinon nous devons choisir une variable x_i non entière dans la solution optimale de la relaxation linéaire pour faire un branchement : l'ensemble des solutions sera divisé en deux, celles pour lequelles $x_i \leq \lfloor x_i^0 \rfloor$ et celles pour lesquelles $x_i \geq \lfloor x_i^0 \rfloor + 1$. Ensuite chaque nouveau programme (correspondant à un sous-problème)

sera évalué. L'opération de branchement ne sera pas effectuée dans l'un des cas suivants :

1. le programme linéaire n'est pas réalisable ;
2. la valeur optimale est supérieure (resp. inférieure) à la valeur de la meilleure solution réalisable trouvée dans le cas d'un problème de minimisation (resp. maximisation) ;
3. la solution est entière, donc réalisable pour le programme linéaire en nombres entiers ; la meilleure solution réalisable est alors mise à jour s'il y a lieu.

L'algorithme s'arrête quand il ne trouve plus de problème à évaluer. L'algorithme construit une arborescence, nommée **arbre de branchement**, constituée d'un ensemble de noeuds. Un noeud qui contient un sous-problème n'ayant pas encore été évalué est dit noeud actif. La figure 2.1 est une schématisation de l'algorithme.

2.4 La méthode des plans coupants

La méthode des plans coupants consiste à résoudre la relaxation linéaire d'un programme linéaire en nombres entiers, et ensuite à partir de la solution obtenue, à essayer d'identifier des contraintes violées par cette solution mais qui sont valides pour le programme linéaire en nombres entiers initial. Ces contraintes sont appelées des coupes (voir la définition 2.2.7). Ensuite, l'algorithme ajoute les coupes trouvées au programme linéaire relaxé résolu au noeud courant et le résout de nouveau. L'ajout d'une coupe au programme permet la suppression d'une partie inutile du polyèdre de la relaxation contenant la solution courante. L'opération d'identification des coupes (ou plans coupants) est appelée **méthode de séparation des inégalités valides**. Cette dernière est répétée jusqu'à ce qu'on trouve une solution entière.

Grötschel, Lovász et Schrijver (1988) ont montré qu'un problème peut être résolu en un temps polynomial si et seulement si la séparation des contraintes définissant le polytope se fait en temps polynomial. En effet, ils ont démontré que l'optimisation sur un polyèdre donné ne dépend pas du nombre de contraintes du système décrivant le polyèdre, mais plutôt du problème de séparation lié à ce système. Ce problème consiste,

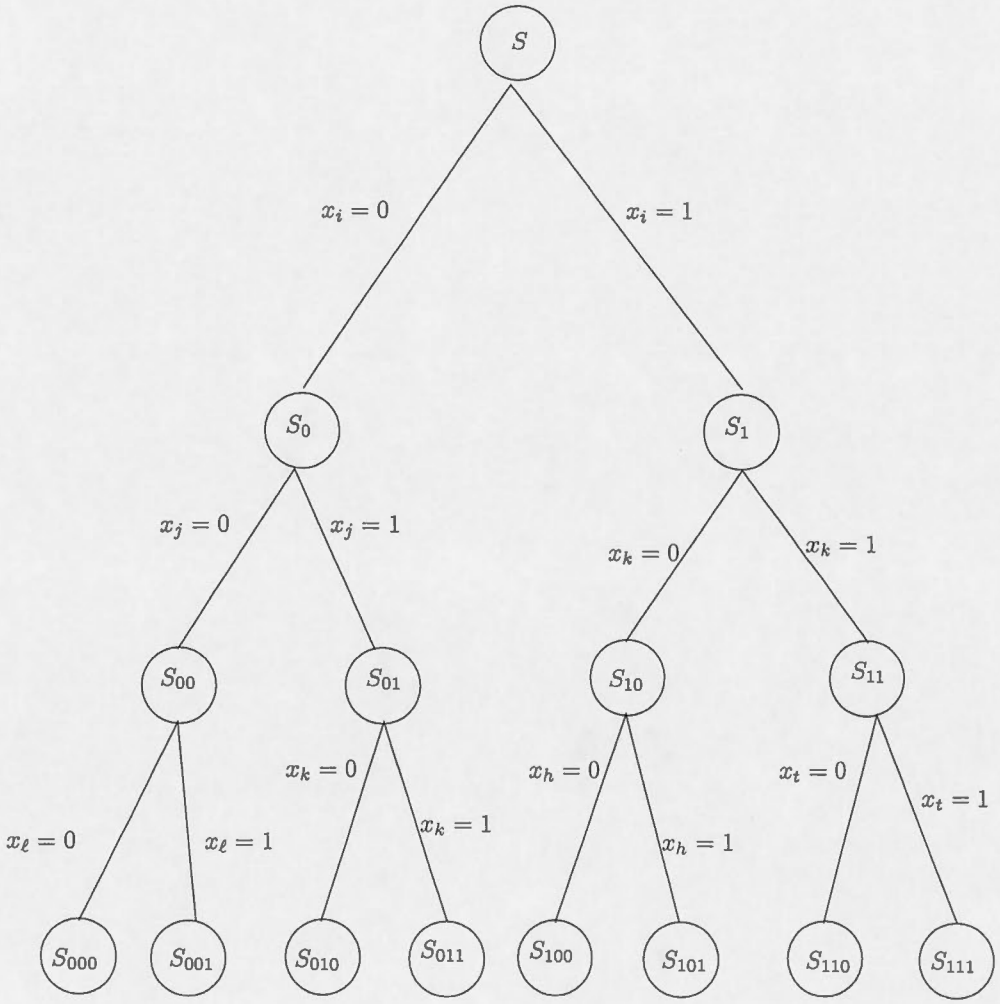

FIGURE 2.1 Arbre de branchement

étant donné une solution x, à déterminer si x vérifie le système, et si elle ne le vérifie pas, à trouver une contrainte qui soit violée par x. Il est peu probable que le problème général de séparation pour le MDVSP puisse être résolu en temps polynomial, puisque ce problème est NP-complet.

L'algorithme général de la méthode des plans coupants est donné ci-dessous.

Algorithme 2.1 Plans coupants

1: Définir D l'ensemble des contraintes du problème tel que $conv(S) \subset D$.
2: Trouver la solution optimale $\hat{x}$ de la relaxation linéaire du problème.
3: **tant que** $\hat{x} \notin conv(S)$ **faire**
4: Trouver une inégalité $dx \geq a$ violée par $\hat{x}$
5: Ajouter $dx \geq a$ au système d'inégalités définissant D
6: Trouver la solution optimale $\hat{x}$
7: **fintant que**

Il est évident que la méthode de séparation des inégalités valides diffère d'un problème à un autre. Notre méthode appliquée au MDVSP sera expliquée dans le chapitre suivant. Nous ne connaissons pas tous les plans coupants pour le MDVSP.

2.5 La méthode « Branch-and-Cut »

2.5.1 Principe de la méthode

Cette approche a été utilisée pour la première fois par Grötschel, Junger et Reinelt (1984) pour le problème de l'ordre linéaire. Le terme de « Branch-and-Cut » fut introduit par Padberg et Rinaldi (1987) pour un algorithme de résolution du problème du voyageur de commerce. Padberg et Rinaldi (1991) ont fourni une première version de l'algorithme de « Branch-and-Cut », en introduisant de nouvelles procédures telles que la génération de colonnes et une procédure de séparation sophistiquée et en utilisant un logiciel pour résoudre des programmes linéaires.

L'algorithme de « Branch-and-Cut » est une méthode qui conjugue la méthode

de séparation et évaluation progressive et celle des plans coupants. Pour résoudre un programme linéaire en nombres entiers, l'algorithme de « Branch-and-cut » commence par relâcher les contraintes d'intégrité et résoudre la relaxation linéaire du problème; puis l'algorithme applique la méthode des plans coupants à la solution trouvée. Si une solution entière n'est toujours pas trouvée, alors le problème est subdivisé en sous-problèmes qui seront résolus de la même manière jusqu'à l'obtention d'une solution entière.

Considérons le programme linéaire en nombres entiers ci-dessous, où toutes les données sont entières.

$$(ILP)\begin{cases} \min\ z = c^T x \\ \quad \text{s.c.} \quad Ax \geq b \\ \qquad\quad x \geq 0, x \in \mathbb{Z}^n \end{cases}$$

Le schéma général de l'algorithme se présente comme suit.

Étape 1 (initialisation) Soit ILP^0 le programme linéaire initial à résoudre et L (initialisée à $\{ILP^0\}$) la liste des noeuds actifs dans l'arbre de branchement. La borne supérieure est $\bar{Z} = +\infty$ et la borne inférieure est $\tilde{Z} = -\infty$.

Étape 2 (test d'arrêt) Si $L = \emptyset$, alors la solution x^*, pour laquelle la valeur de l'objectif est $\bar{Z}$, est la solution optimale. Si un tel x^* n'existe pas (c'est-à-dire que $\bar{Z} = +\infty$), alors ILP^0 est non-réalisable.

Étape 3 (sélection du problème) On sélectionne un problème ILP^k de L et on le retire de L.

Étape 4 (relaxation) On résout la relaxation linéaire de ILP^k. Si la relaxation est non réalisable, on pose $\tilde{Z}_k = +\infty$ et on effectue l'étape 6. Soit $\tilde{Z}_k$ la valeur optimale de la fonction-objectif de la relaxation. Si elle est finie, dénotons x^{kR} la solution optimale. Sinon on pose $\tilde{Z}_k = -\infty$.

Étape 5 (ajout des plans coupants) On cherche les plans coupants des familles

connues qui sont violés par la solution x^{kR} ; s'ils existent, on les ajoute à la relaxation et on va à l'étape 4.

Étape 6 (évaluation et élimination)

(a) Si $\tilde{Z}_k > \bar{Z}$, aller à l'étape 2 ;

(b) Si $\tilde{Z}_k < \bar{Z}$ et x^{kR} est une solution réalisable et entière, alors poser $\bar{Z} = \tilde{Z}_k$ et retirer de L tout problème ILP^{ℓ} avec $\tilde{Z}_{\ell} \geq \bar{Z}$ et aller à l'étape 2 ;

Étape 7 (branchement) On crée des noeuds fils du noeud courant, on les ajoute à la liste des noeuds actifs ; la borne inférieure de chaque noeud fils est initialement égale à celle du noeud courant, aller à l'étape 2.

2.5.2 Exemple d'application

L'exemple suivant est tiré de la référence (Mitchell, 1999) et est illustré par la figure 2.2. Considérons le programme linéaire en nombres entiers ci-dessous.

$$(P)\begin{cases} \min\ z = -5x_1 - 6x_2 \\ \text{s.c.} \quad x_1 + 2x_2 \leq 7 \\ \qquad\quad 2x_1 - x_2 \leq 3 \\ \qquad\quad x_1, x_2 \geq 0, \text{entiers.} \end{cases}$$

La relaxation de (P), dénotée (P'), est obtenue en relâchant les contraintes d'intégrité. Nous appliquons la méthode de « Branch-and-Cut » pour résoudre (P). La première étape consiste à résoudre (P'). La solution optimale de (P') est $x^1 = (2.6, 2.2)$ et la valeur de l'objectif est $z_1 = -26.2$. Nous avons deux choix : ou bien améliorer la solution de (P') en ajoutant un plan coupant, par exemple $x_1 + x_2 \leq 4$, ou bien décomposer (P') en deux sous-problèmes en subdivisant le domaine de l'une des variables. Supposons que l'algorithme opte pour le deuxième choix. La variable choisie est x_2 et deux sous-problèmes sont obtenus :

$$(P_0)\begin{cases} \min\ z = -5x_1 - 6x_2 \\ \text{s.c.} \quad x_1 + 2x_2 \leq 7 \\ \qquad\quad 2x_1 - x_2 \leq 3 \\ \qquad\quad x_2 \geq 3 \\ \qquad\quad x_1, x_2 \geq 0, \text{entiers.} \end{cases}$$

et

$$(P_1)\begin{cases} \min\ z = -5x_1 - 6x_2 \\ \text{s.c.} \quad x_1 + 2x_2 \leq 7 \\ \qquad\quad 2x_1 - x_2 \leq 3 \\ \qquad\quad x_2 \leq 2 \\ \qquad\quad x_1, x_2 \geq 0, \text{entiers.} \end{cases}$$

La solution optimale de la relaxation linéaire de (P_0) est $x^2 = (1,3)$ et la valeur optimale est $z_2 = -23$. La solution optimale est entière; ainsi (P_0) est résolu, et cette solution est la meilleure solution réalisable de (P) trouvée jusqu'ici. La solution optimale de la relaxation linéaire de (P_1) est $x^3 = (2.5, 2)$, avec un optimum de $z_3 = -24.5$; cette solution n'est pas entière et donc (P_1) n'est pas encore résolu. Supposons qu'une approche de plans coupants est utilisée et qu'on ajoute l'inéquation $x_1 + 2x_2 \leq 6$ à (P_1). Cette inéquation est valide car $2x_1 - x_2 \leq 3$ et $x_2 \leq 2$ impliquent que $2x_1 \leq 5$ $=>$ $x_1 \leq 5/2$ $=>$ $x_1 \leq 2$ (car x_1 est entier). De plus,

$$x_2 \leq 2 \ => 2x_2 \leq 4,$$

d'où $x_1 + 2x_2 \leq 6$. Cette dernière inégalité est violée par la solution $x^3 = (2.5, 2)$, donc c'est une coupe. Le sous-problème obtenu est

$$(P_{10}) \begin{cases} \min\ z = -5x_1 - 6x_2 \\ \text{s.c.} \quad x_1 + 2x_2 \leq 7 \\ \qquad\quad 2x_1 - x_2 \leq 3 \\ \qquad\quad x_2 \leq 2 \\ \qquad\quad x_1 + 2x_2 \leq 6 \\ \qquad\quad x_1, x_2 \geq 0, \text{entiers.} \end{cases}$$

La solution optimale de la relaxation de (P_{10}) est $x^4 = (2.4, 1.8)$ et la valeur de l'optimum est $z_4 = -22.6$. La solution courante n'est toujours pas entière ; par contre, la valeur courante de l'objectif est supérieure à la valeur de la meilleure solution réalisable courante de (P) $(z_4 > z_2)$. Par conséquent, il est inutile de continuer le branchement au noeud (P_{10}). Comme il n'existe pas d'autres sous-problèmes à résoudre, la solution optimale de (P) est la meilleure solution réalisable trouvée, c'est-à-dire $x^2 = (1, 3)$, avec une valeur de -23.

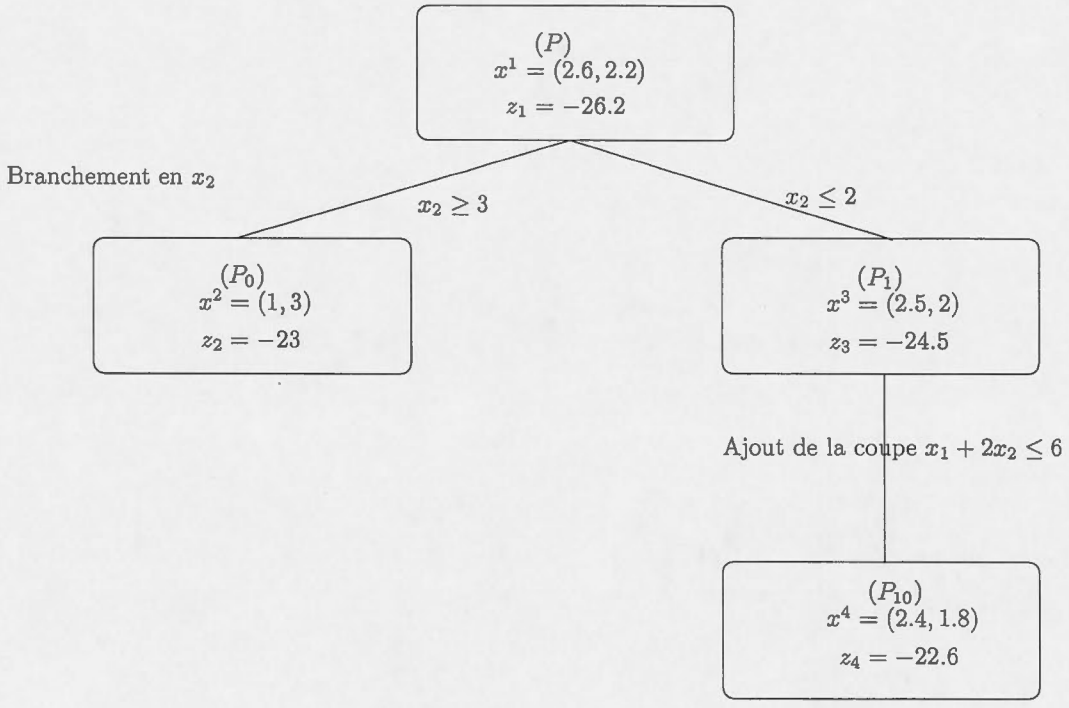

FIGURE 2.2 Exemple d'application de l'algorithme de « Branch-and-Cut »

CHAPITRE III

MÉTHODE DE DÉTERMINATION DES INÉGALITÉS VALIDES POUR LE MDVSP

3.1 Introduction

Dans le présent chapitre, nous présentons la formulation du problème du MDVSP, ainsi que quelques définitions et propositions qui nous permettent la compréhension des méthodes présentées. La section 3.5 détaille un algorithme de type « Branch-and-Cut » pour le problème du stable, proposé par Nemhauser et Sigismondi (1992). Nous expliquons dans la section 3.6 la relation entre l'algorithme présenté dans la section 3.5 et la méthode de détermination des inégalités valides pour le MDVSP que nous proposons. Nous présentons les détails et les algorithmes de chaque étape de cette méthode. Nous terminons par une description de l'algorithme de séparation des inégalités valides du MDVSP, ainsi que de l'algorithme de résolution du problème.

3.2 Formulation du problème

La formulation que nous avons adoptée est celle utilisée par Hadjar, Marcotte et Soumis (2006). Considérons un graphe $G = (V, A)$ tel que V représente l'ensemble des sommets et A représente l'ensemble des arcs. L'ensemble des sommets V est l'union de l'ensemble des tâches $\{T_1, T_2, ..., T_n\}$ et de celui des dépôts $\{D_1, D_2, ..., D_{|K|}\}$, l'ensemble des arcs, A, contient tous les arcs de la forme (D_k, T_i) ou (T_i, D_k) pour tout $k \in K$ et $i = 1, 2, \ldots, n$ et $|K|$ copies des arcs de la forme (T_i, T_j) (un arc pour chaque couleur).

Afin de simplifier, nous affectons à chaque dépôt une couleur. Nous utilisons le triplet (i, j, k) pour faire référence à un arc qui débute à T_i et se termine à T_j et qui est de couleur k $(i, j \leq n)$. Notons que si $i = n + k$ et $j \leq n$, alors (i, j, k) représente l'arc (D_k, T_j).

Définition 3.2.1

- $\delta^+(i) = \{j \mid (i, j, k) \in A\}$
- $\delta^-(i) = \{j \mid (j, i, k) \in A\}$
- $\delta(i) = \delta^+(i) \cup \delta^-(i)$

Définition 3.2.2

Un cycle monochromatique est un cycle dont tous les arcs ont la même couleur.

Le problème du MDVSP peut être vu comme un problème de partitionnement des tâches de G en cycles monochromatiques. Notons que le nombre de cycles incluant D_k ne dépasse pas v_k. La formulation du problème sous forme de programme linéaire nécessite la définition des variables, des contraintes à respecter et de l'objectif à atteindre.

Nous avons besoin d'une variable binaire par arc.

$$X_{ij}^k = \begin{cases} 1 & \text{si } (i, j, k) \text{ appartient à un chemin (ou cycle) dans une affectation réalisable} \\ 0 & \text{sinon} \end{cases}$$

Les contraintes sont les suivantes :

- chaque tâche est effectuée par un seul véhicule,
- le nombre de véhicules quittant D_k est au plus égal à v_k, et
- la contrainte de conservation de flot doit être satisfaite par chaque tâche et chaque dépôt.

L'objectif cherché est de minimiser le coût total d'une affectation réalisable.

Le problème se présente alors sous forme de programme linéaire en nombres entiers.

$$
(P)\begin{cases}
\min \quad \sum_{(i,j,k)\in A} c_{ij} X_{ij}^k & \\
\text{s.c.} \quad \sum_{k\in K} \sum_{j\in\delta^+(i)} X_{ij}^k = 1 & i = 1,\ldots,n \\
\sum_{j=1}^{n} X_{n+k,j}^k \leq v_k & \forall k \in K \\
\sum_{j\in\delta^-(i)} X_{ji}^k - \sum_{j\in\delta^+(i)} X_{ij}^k = 0 & \forall k \in K \text{ et } i = 1,\ldots,n,n+k \\
X_{ij}^k \in \{0,1\} & \forall (i,j,k) \in A
\end{cases}
$$

3.3 Les inégalités valides pour le MDVSP

Bertossi, Carraresi et Gallo (1987) ont montré qu'il existe une relation entre le MDVSP et le problème du couplage. La différence entre les deux problèmes réside dans la notion de conflit : dans le problème du couplage deux arcs adjacents sont toujours en conflit, alors que ceci n'est pas nécessairement vrai dans le MDVSP. Malgré cela, il est possible d'utiliser le principe des inégalités valides introduites dans le problème du couplage.

Hadjar, Marcotte et Soumis (2006) ont observé que si la taille des instances du MDVSP est petite (c'est-à-dire est de 50 tâches ou moins), la solution de la relaxation linéaire du MDVSP est généralement entière. D'autre part, si elle n'est pas entière, les flux des variables sont souvent des valeurs multiples de 1/2, ce qui sont associées, en optimisation combinatoire, aux cycles impairs.

Nous avons besoin de certaines définitions et propositions introduites par Hadjar, Marcotte et Soumis (2006).

3.4 Quelques définitions et propositions

Définition 3.4.1

Soit (i,j,k) et (i',j',k') deux arcs adjacents dans le multigraphe G qui ne sont pas incidents aux dépôts (c'est-à-dire que $i,j,i',j' \leq n$). On dit qu'ils sont en conflit si $k \neq k'$ ou $i = i'$ ou $j = j'$.

Définition 3.4.2

On dit que le sous-multigraphe partiel $G' = (S,F)$ de G est conflictuel si :

- *$|S|$ est impair,*
- *S ne contient aucun dépôt,*
- *quels que soient deux arcs adjacents dans F, ils sont en conflit.*

On remarque que quels que soient les arcs (i,j,k) et (i',j',k') en conflit, l'inégalité $X_{ij}^k + X_{i'j'}^{k'} \leq 1$ est valide pour (P) (le problème initial). Cette inégalité est vérifiée au niveau de chaque sommet de G', puisque selon la définition précédente quels que soient deux arcs adjacents dans F, ils sont en conflit. Par conséquent, la somme au niveau de chaque sommet du sous-multigraphe G' de l'inégalité précédente implique que $2 \sum_{(i,j,k)\in F} X_{ij}^k \leq |S|$ est vérifiée. Il s'ensuit que, étant donné un sous-multigraphe conflictuel $G' = (S,F)$, l'inégalité $\sum_{(i,j,k)\in F} X_{ij}^k \leq \left\lfloor \frac{|S|}{2} \right\rfloor$ est valide pour (P).

Malheureusement, toute solution fractionnaire ne contient pas un sous-multigraphe conflictuel, mais il est possible d'avoir une structure semblable à celle illustrée dans l'exemple de la figure 3.1, où les différents types de lignes représentent des couleurs. Par exemple, si nous considérons la solution fractionnaire de la figure 3.1, nous remarquons que (T_1,T_4) et (T_4,T_3) ne sont pas en conflit car T_4 est l'extrémité finale du premier et l'extrémité initiale du second et les arcs sont de même couleur. Par contre, (T_5,T_4) et (T_1,T_4) ainsi que (T_5,T_4) et (T_4,T_3) sont en conflit parce qu'ils ne sont pas de même couleur. Si nous considérons les arcs (T_1,T_2), (T_3,T_2), (T_4,T_3), (T_1,T_4) et (T_5,T_4), l'in-

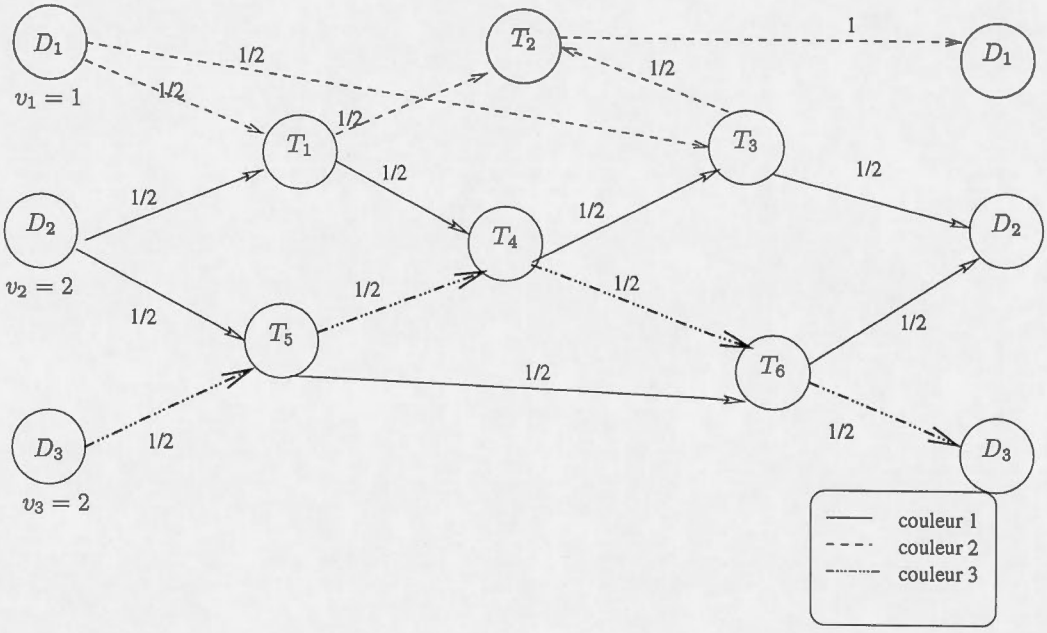

FIGURE 3.1 Exemple de solution qui contient un cycle épineux

égalité $X^1_{12} + X^1_{32} + X^2_{14} + X^2_{43} + X^3_{54} \leq 2$ élimine la solution fractionnaire dans ce cas ; cette inégalité est donc valide pour le problème (P). L'arc (T_5, T_4) qui crée le conflit au niveau du sommet T_4 est nommé épine et l'ensemble des arcs du plan coupant est appelé cycle épineux.

Remarque 3.4.3

L'exemple 3.1 est tiré de l'article de Hadjar, Marcotte et Soumis (2006).

Définition 3.4.4

Soit C un cycle de G. On dit que C est un cycle épineux si l'ensemble de ses sommets peut être partitionné en deux ensembles S_1 et S_2 et l'ensemble de ses arcs en F_1 et F_2 tels que :

- *pour chaque sommet $i \in S_1$, deux arcs incidents à i sont en conflit, et*
- *pour chaque sommet $i \in S_2$, il existe un arc (appelé épine) dont l'extrémité finale est i et qui est en conflit avec chacun des arcs incidents à i dans le cycle C.*

Afin de généraliser la notion de cycle épineux, nous avons besoin de la définition suivante.

Définition 3.4.5

Pour tout sous-ensemble A' de A, on définit

- *$Col(A')$ comme $\{ k \in K \mid A'$ contient au moins un arc de couleur $k \}$,*
- *$\delta^-_{A'}(i)$ (resp. $\delta^+_{A'}(i)$) comme l'ensemble des arcs appartenant à A' tels que la destination de l'arc (resp. l'origine de l'arc) est le sommet i et*
- *$\delta_{A'}(i) = \delta^-_{A'}(i) \cup \delta^+_{A'}(i)$.*

Définition 3.4.6

Soit $G' = (S, F)$ un sous-multigraphe partiel de G. On dit que G' est un sous-multigraphe épineux, si S peut être partitionné en S_1, S_2 et S_3 et F peut être partitionné en F_1 et F_2 de telle sorte que :

- *$S_1 \cup S_2$ ne contient aucun dépôt,*
- *$|S_1|$ est impair et au moins égal à 3,*
- *$\delta^-_{F_2}(i) = \emptyset$ et $Col(\delta^-_{F_1}(i)) \bigcap Col(\delta^+_{F_1}(i)) = \emptyset$ pour tout $i \in S_1$,*
- *$Col(\delta^-_{F_2}(i)) \bigcap Col(\delta^-_{F_1}(i)) = \emptyset$ et $Col(\delta^-_{F_2}(i)) \bigcap Col(\delta^+_{F_1}(i)) = \emptyset$ pour tout $i \in S_2$, et*
- *$\delta^-_F(i) = \emptyset$, $\delta^+_F(i) \subseteq F_2$ et $\delta^+_F(i) \neq \emptyset$ pour tout $i \in S_3$.*

Proposition 3.4.7

Soit $G' = (S, F)$ un sous-multigraphe épineux de G, où $S = S_1 \cup S_2 \cup S_3$ et $F = F_1 \cup F_2$ et les S_i et les F_j satisfont la définition précédente. Alors l'inégalité $\sum_{(i,j,k)\in F} X^k_{i,j} \leq \left\lfloor \frac{|S_1|}{2} \right\rfloor + |S_2|$ est valide pour (P).

Définition 3.4.8

Une corde est une arête qui joint deux sommets d'un cycle mais qui n'appartient pas au cycle.

Définition 3.4.9

Un trou est un cyle impair sans corde.

3.5 Un algorithme de type « Branch-and-Cut » pour le problème du stable

Soit un graphe $G = (V, E)$ sans boucles ni arêtes multiples, où V est l'ensemble des sommets et E représente l'ensemble des arêtes de G. Un stable de G est un sous-

ensemble de sommets deux à deux non adjacents ; en d'autres termes chaque arête du graphe a au plus une extrémité dans le stable. Soit p le vecteur des poids associés aux sommets de G ; le poids d'un sous-ensemble de sommets est la somme des poids de tous ses sommets. Le problème du stable de poids maximum consiste à trouver un stable S tel que la somme des poids des sommets de S soit maximale.

Il est facile de formuler mathématiquement le problème. Soit la variable binaire x_v telle que

$$x_v = \begin{cases} 1 & \text{si le sommet } v \text{ appartient au stable} \\ 0 & \text{sinon.} \end{cases}$$

Le problème du stable s'écrit sous forme de programme linéaire en nombres entiers comme suit :

$$(Stable) \begin{cases} \max \quad z = \sum_{v \in V} p_v x_v \\ \text{s.c.} \qquad x_u + x_v \leq 1 \quad \forall uv \in E \\ \qquad\qquad x_v \in \{0,1\} \quad \forall v \in V. \end{cases}$$

Nemhauser et Sigismondi (Nemhauser et *al.*, 1992) ont présenté un algorithme pour la résolution du problème du stable de poids maximum. Les plans coupants sont obtenus par les inégalités de cliques et les inégalités des trous impairs liftées (un trou est un cycle sans corde). La procédure de détermination des inégalités valides correspondant aux trous impairs est la suivante.

Soit $G = (V, E)$ un graphe non orienté, où V représente l'ensemble des sommets et E l'ensemble des arêtes. La solution optimale fractionnaire de la relaxation linéaire du problème traité est dénotée $x = (x_i)_{i \in V}$. À chaque arête $ij \in E$, nous associons le coût $c_{ij} = 1 - x_i - x_j \geqslant 0$. Nous construisons un graphe biparti $G' = (V, V', E')$, où G'

contient les arêtes ij' et ji' si et seulement si ij est une arête de G. Les coûts associés à chaque arête sont $c_{ij'}$ et $c_{ji'}$, où $c_{ij'} = c_{ji'} = c_{ij}$. Notons qu'un chemin impair dans G' de v_0 à v'_0 est de la forme $(v_0, v'_1, v_2, v'_3, ..., v_{2k}, v'_0)$, correspondant au cycle impair $(v_0, v_1, v_2, v_3, ..., v_{2k}, v_0)$ dans G. Ce dernier est un cycle comprenant $(2k+1)$ sommets. Les poids des chemins dans G et dans G' sont identiques et égaux à

$$2k + 1 - 2\sum_{j=0}^{2k} x_{v_j}. \tag{3.1}$$

Il suit que le chemin de poids minimum de v_0 à v'_0 fournit un cycle impair C dans G qui maximise $\sum_{v \in C} x_v - \frac{|C|-1}{2}$ parmi tous les cycles impairs de G contenant v_0, et

$$\sum_{v \in C} x_v - \frac{|C|-1}{2} > 0 \tag{3.2}$$

si et seulement si le poids du chemin correspondant dans G' est inférieur à 1 (c'est-à-dire $2k + 1 - 2\sum_{j=0}^{2k} x_{v_j} < 1$).

Supposons que nous avons trouvé un cycle impair de poids minimum C contenant v_0 et tel que $\sum_{v \in C} x_v > \frac{|C|-1}{2}$. Trois cas sont possibles :

- si $|C| = 3$, le cycle C est un triangle et nous avons obtenu une contrainte violée,
- si $|C| \geq 5$ et C est un trou ; nous avons une contrainte violée aussi,
- si $|C| \geq 5$ et C est un cycle avec une ou plusieurs cordes. on remarque que chaque corde partitionne le cycle C en un cycle impair C_1 et un chemin P avec un nombre pair de sommets. Il suit que $\sum_{v \in C_1} x_v + \sum_{v \in P} x_v = \sum_{v \in C} x_v > \frac{|C|-1}{2} = \frac{|C_1|-1}{2} + \frac{|P|}{2}$.

  Dans le problème du stable, nous avons les contraintes sur les arêtes $x_u + x_v \leq 1$, pour tout $uv \in E$. Si nous faisons la somme des contraintes sur les arêtes pour une famille $\mathcal{F}$ d'arêtes partitionnant P nous obtenons

$$\sum_{vu \in \mathcal{F}} (x_u + x_v) \leq \frac{|P|}{2} \tag{3.3}$$

$$\Longrightarrow \sum_{v \in P} x_v \leq \frac{|P|}{2} \iff \frac{|P|}{2} - \sum_{v \in P} x_v \geq 0, \tag{3.4}$$

et donc $\sum_{v \in C_1} x_v > \frac{|C_1|-1}{2} + \frac{|P|}{2} - \sum_{v \in P} x_v > \frac{|C_1|-1}{2}$.

Nous avons obtenu un cycle impair C_1 qui satisfait la condition (3.2). Nous appliquons la même procédure jusqu'à ce que nous obtenions ou bien un triangle ou bien un trou impair. L'inégalité correspondant à C est redondante si C contient une corde.

3.6 Méthode de détermination des inégalités valides pour le MDVSP

La méthode de détermination des inégalités valides pour le MDVSP que nous proposons consiste à commencer par la résolution de la relaxation linéaire du problème (P) présenté dans la section 3.2. Si la solution obtenue n'est pas entière, alors nous l'utilisons pour construire un sous-multigraphe partiel orienté, $H = (S, F)$; ensuite, nous associons à H un graphe adjoint non orienté $L(H)$ (sous certaines conditions). À ce dernier nous appliquons la méthode de Nemhauser et Sigismondi avec quelques modifications, afin de détecter les trous impairs qui correspondent à des cycles impairs conflictuels (épineux) ou des sous-multigraphes conflictuels (épineux) dans H. Nous montrerons que les cycles impairs conflictuels (épineux) nous fournissent des inégalités valides, et afin d'enrichir nos inégalités valides, nous décrirons une procédure de liftage. Dans cette section nous présenterons les détails de chaque étape.

3.6.1 Création des graphes

Définition 3.6.1

Soit $L(H) = (F, L)$ le graphe adjoint non orienté associé au sous-multigraphe partiel orienté $H = (S, F)$ (le support de la solution de la relaxation linéaire). $L(H)$ est

défini comme suit :

- *les sommets de $L(H)$ sont les arêtes de H,*
- *deux sommets de $L(H)$ sont reliés par une arête si et seulement si les deux arcs sont en conflit dans H,*
- *x_i est le poids du sommet $i \in V$, qui est égal au flux de l'arc représenté par le sommet i (c'est-à-dire que si i est un sommet de $L(H)$ qui représente l'arc (ℓ, j, k) de H, alors $x_i = X^k_{\ell j}$), et*
- *le poids de chaque arête de $L(H)$ est égal à $1 - x_i - x_j$.*

Remarque 3.6.2

Le poids de chaque arête est positif ou nul, puisque par la remarque suivant la définition 3.4.2, quels que soient les deux arcs (i, j, k) et (i', j', k') en conflit, l'inégalité $X^k_{ij} + X^{k'}_{i'j'} \leq 1$ est valide, d'où suit la relation $1 - X^k_{ij} - X^{k'}_{i'j'} \geq 0$.

Définition 3.6.3

Soit $B(H) = (F', F'', L')$ un graphe biparti non orienté, construit à partir de $L(H) = (F, L)$, tel que :

- *F' contient les sommets de $L(H)$ ($F' = F$),*
- *F'' contient une copie des sommets de F',*
- *$L' = \{\{i, copie(j)\} |\ ij \in L\} \cup \{j, copie(i)\} |\ ij \in L\}$,*
- *x_i (resp. $x_{copie(i)}$) est le poids du sommet $i \in F'$ (resp. $copie(i) \in F''$), avec $x_i = x_{copie(i)} = x_\ell$ où ℓ est le sommet correspondant dans le graphe $L(H)$, pour $i \in F'$ et $copie(i) \in F''$, et*
- *le poids de chaque arête $\{i, copie(j)\} \in L'$ est égal à $1 - x_i - x_j$.*

Exemple

L'exemple présenté dans la figure 3.1 permet de construire les graphes H, $L(H)$ et $B(H)$ qui sont présentés dans les figures 3.2, 3.3 et 3.4. Notons que dans cet exemple,

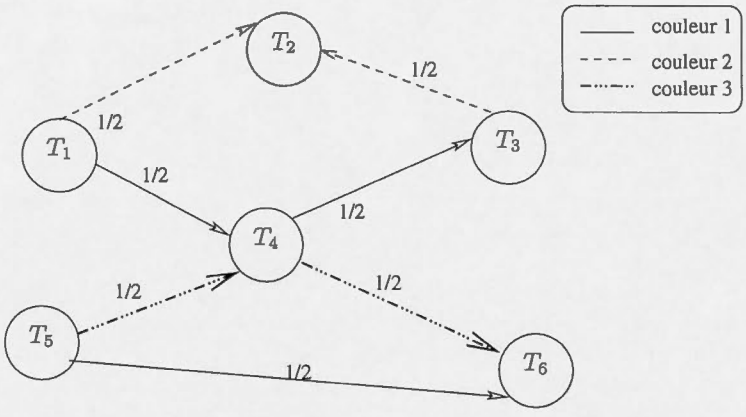

FIGURE 3.2 Exemple d'un graphe H

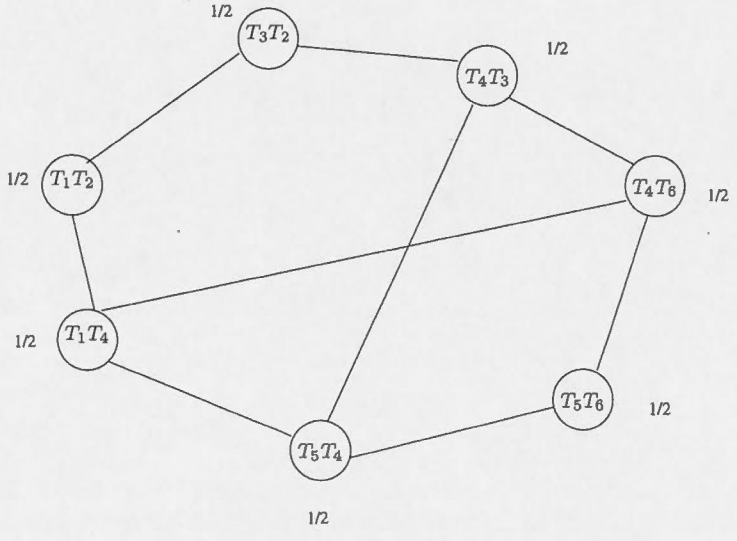

FIGURE 3.3 Graphe $L(H)$ correspondant au graphe H de la figure 3.2

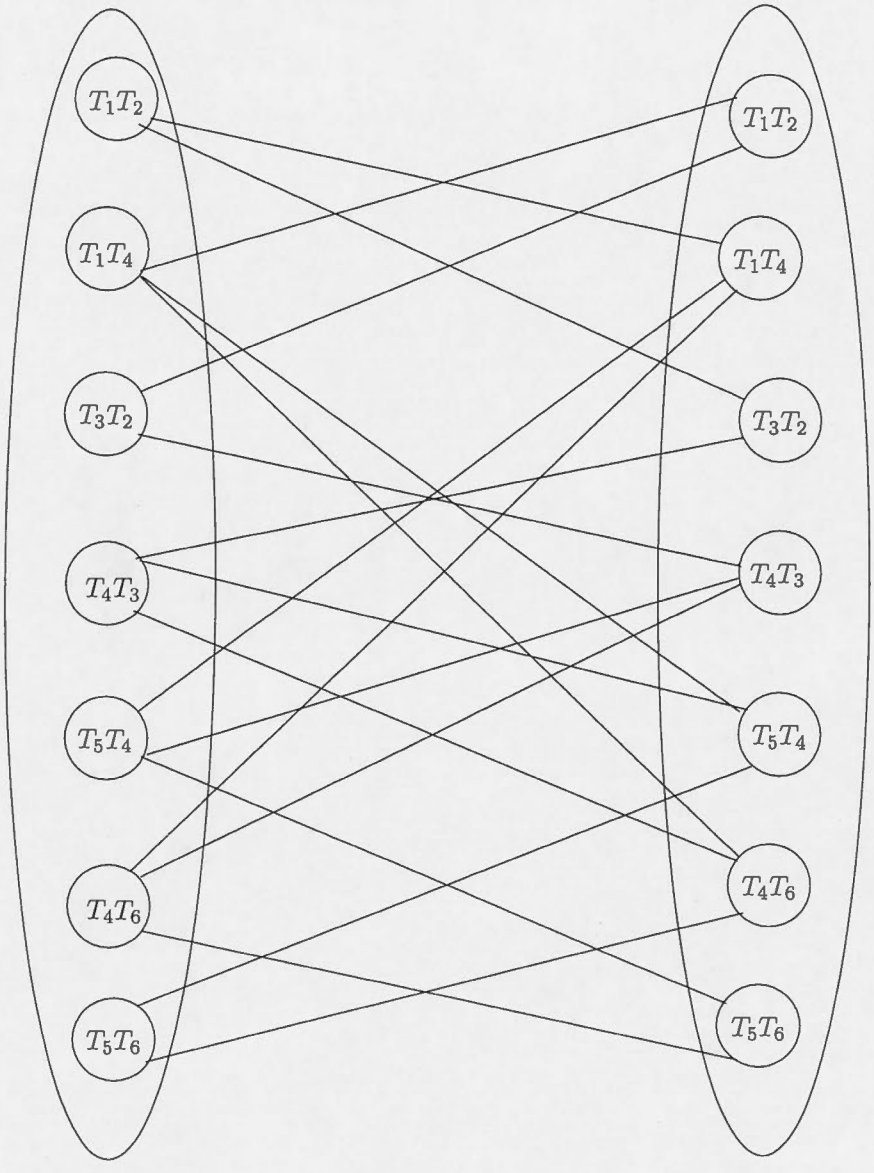

FIGURE 3.4 Graphe $B(H)$ correspondant au graphe $L(H)$ de la figure 3.3

seul T_4 est un sommet épineux.

3.6.2 Recherche du plus court chemin

Le meilleur algorithme pour résoudre le problème du plus court chemin est l'algorithme de Dijkstra, élaboré en 1959, qui trouve le plus court chemin d'un sommet source à tous les autres sommets d'un graphe orienté et pondéré $G = (V, A)$, dans le cas où les poids de tous les arcs du graphe sont positifs ou nuls. L'algorithme de Dijkstra gère un ensemble de sommets S, dont les longueurs finales des plus courts chemins à partir d'une source unique ont été calculées. Soit s la source et $w(u, v) \geq 0$ le poids de l'arc (u, v) ; $w : V \rightarrow \mathbb{R}$ est la fonction de pondération associée au graphe G. À chaque itération, l'algorithme 3.1 (tiré de Cormen et *al.*) choisit un sommet u de V qui n'est pas dans S et dont la distance présumée à la source est minimale, ajoute u à S, ensuite relâche tous les arcs adjacents à u. Le rôle de la procédure *Relâcher*(u, v, w) (voir l'algorithme 3.2) consiste à tester s'il est possible d'améliorer le plus court chemin vers v, en passant par u. Si c'est vrai, $d(v)$, qui représente la longueur du plus court chemin de la source à v trouvé jusqu'ici, est actualisée, ainsi que le prédécesseur de v qui deviendra u.

Algorithme 3.1 Dijkstra(G, w, s)

1: Détermination de la source s
2: $S := \emptyset$
3: $d(s) = 0, d(u) = +\infty$ pour tout $u \neq s$
4: $F := V$ (Les éléments de la file de priorité F sont identifiés par les valeurs de d(.))
5: **tant que** $F \neq \emptyset$ **faire**
6: $\quad min := Extraire\_Min(F)$
7: $\quad S := S \cup \{min\}$
8: $\quad$ **pour** *chaque sommet v adjacent à min* **faire**
9: $\quad\quad$ *Relâcher(min, v, w)*
10: $\quad$ **finpour**
11: **fintant que**

Algorithme 3.2 *Relâcher(u,v,w)*

1: **si** $d(v) > d(u) + w(u,v)$ **alors**
2: $d(v) := d(u) + w(u,v)$
3: *Prédécesseur(v)* $:= u$
4: **finsi**

L'algorithme de Dijkstra fait appel à l'opération *Insérer* (insertion des sommets de V dans la file F) (voir Cormen et *al*) une seule fois pour chaque sommet ; l'opération *Extraire_Min* (voir Cormen et *al*) est aussi appelée une fois par sommet. Par contre, l'opération *Diminuer_ Clé* (voir Cormen et *al*) (implicitement dans *Relâcher(u,v,w)*) est appelée une fois pour chaque arc de la liste d'adjacence de chaque sommet ; donc il y a au plus $|A|$ appels de *Diminuer_ Clé*.

Nous avons donc un coût total de $|V|$*Insérer* + $|V|$*Extraire_Min* + $|A|$*Diminuer_ Clé*. La complexité de l'algorithme dépend de l'implémentation de la file de priorité.

Nous avons implémenté l'algorithme en utilisant les tas binaires, où chaque opération s'exécute en $O(\log(|V|))$. Donc l'agorithme de Dijkstra s'exécute en un temps dans $O((|A|+|V|)\log(|V|))$. Notons aussi qu'il est possible d'utiliser les tas de Fibonacci qui permettent d'obtenir un temps d'exécution en $O(|A|\log(|A|)+|V|)$.

3.6.3 Identification des cycles conflictuels (épineux)

Notre méthode de séparation des plans coupants nous permet de trouver les inégalités valides identifiées par Hadjar, Marcotte et Soumis (2006) ainsi que d'autres inégalités valides. Un trou impair de $L(H)$ correspond soit à un cycle conflictuel (épineux) (voir la figure 3.5) soit à un sous-multigraphe conflictuel (épineux) avec un ou plusieurs sommets d'articulation (voir figure 3.6). Notons qu'un sommet d'articulation est un sommet qui possède deux arcs entrants et deux arcs sortants, tous de même couleur. Notons que, s'il existe un sommet d'articulation dans un sous-multigraphe conflictuel (épineux), alors ce sommet appartient à S_2 et les arcs incidents sont des arcs

appartenants à F_1. Ce cas ne pose aucun problème au niveau du liftage, car la procédure que nous utilisons considère les arcs deux à deux en conflit. Au niveau du point d'articulation, les deux arcs entrants sont en conflit ainsi que les deux arcs sortants. Notons aussi que dans notre étude une épine peut avoir comme origine le sommet épineux. Nous gardons la même définition des cycles conflictuels, mais nous allons redéfinir les cycles épineux ainsi que les sous-multigraphes épineux afin d'introduire la nouvelle définition d'épine.

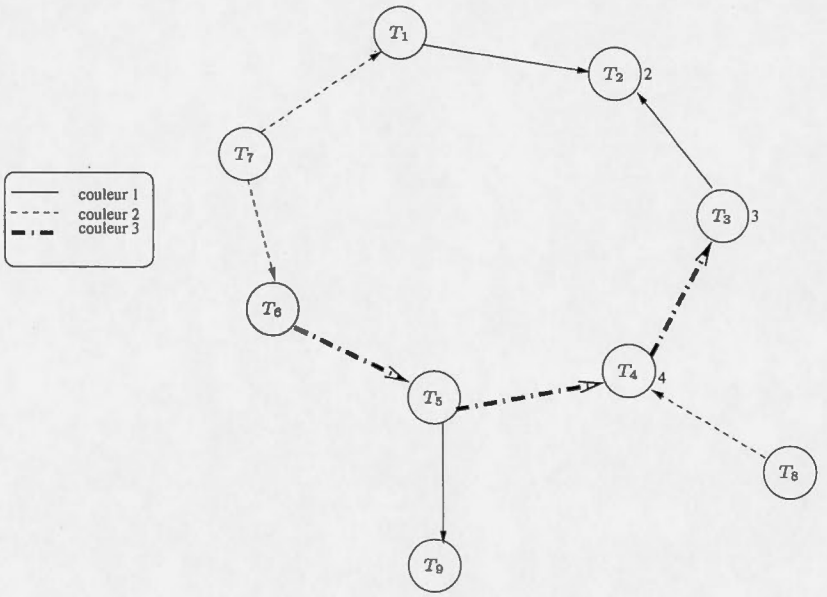

FIGURE 3.5 Exemple de cycle conflictuel épineux correspondant à un trou impair de $L(H)$

Définition 3.6.4

Un sous-multigraphe partiel C de H est dit cycle épineux si son ensemble de sommets peut être partitionné en trois ensembles S_1, S_2 et S_3 et son ensemble d'arcs en deux ensembles F_1 et F_2, de telle sorte que

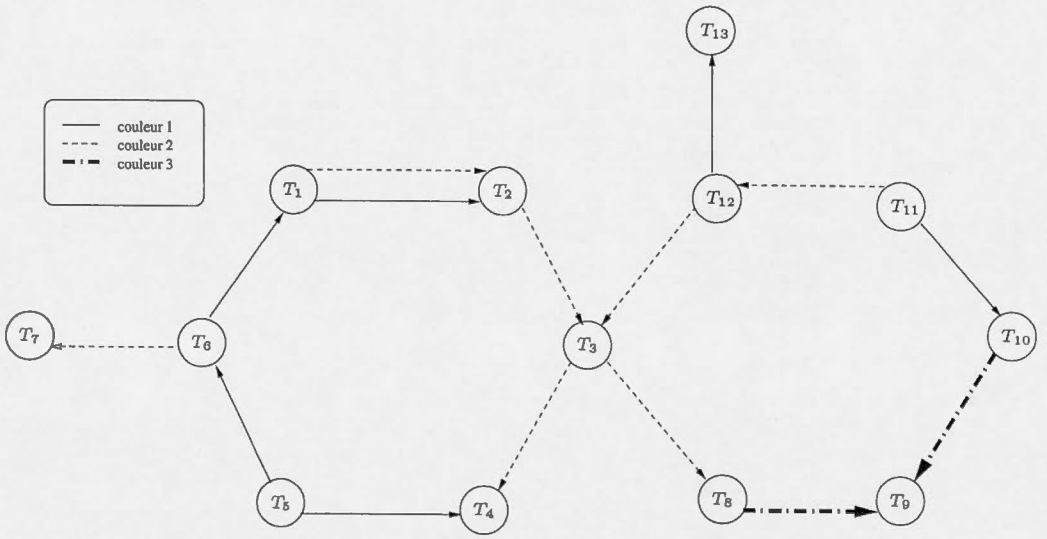

FIGURE 3.6 Exemple de cycle conflictuel épineux avec un point d'articulation correspondant à un trou impair de $L(H)$

- *le sous-multigraphe de C induit par $S_1 \cup S_2$ est un cycle élémentaire dont tous les arcs appartiennent à F_1,*
- *pour chaque sommet $i \in S_1$, les deux arcs incidents à i sont en conflit, et*
- *pour chaque sommet $i \in S_2$, il existe un arc (appelé épine et appartenant à F_2) dont l'une des deux extrémités est le sommet i et qui est en conflit avec chacun des arcs incidents à i dans le cycle C.*

Remarque 3.6.5

Reprenons l'exemple de la figure 3.1, avec la définition précédente. L'arc (T_4, T_6) est aussi une épine et l'ensemble des arcs (T_1, T_4), (T_1, T_2), (T_3, T_2), (T_4, T_3) et (T_4, T_6) est un cycle épineux.

Définition 3.6.6

Soit $H' = (S, F)$ un sous-multigraphe partiel de H. On dit que H' est un sous-multigraphe épineux si S peut être partitionné en S_1, S_2 et S_3 et F peut être partitionné en F_1 et F_2 de telle sorte que :

- *$S_1 \cup S_2$ ne contient aucun dépôt,*
- *$|S_1|$ est impair et au moins égal à 3,*
- *$\delta^-_{F_2}(i) = \emptyset$ et $Col(\delta^-_{F_1}(i)) \bigcap Col(\delta^+_{F_1}(i)) = \emptyset$ pour tout $i \in S_1$,*
- *$Col(\delta^-_{F_2}(i)) \bigcap Col(\delta^-_{F_1}(i)) = \emptyset$ et $Col(\delta^-_{F_2}(i)) \bigcap Col(\delta^+_{F_1}(i)) = \emptyset$ pour tout $i \in S_2$, et*
- *$\delta^-_F(i) = \emptyset$ ou $\delta^+_F(i) = \emptyset$, $\delta_F(i) \subseteq F_2$ pour tout $i \in S_3$.*

Proposition 3.6.7

Tout trou impair C de $L(H)$ correspond à un cycle conflictuel (épineux) ou un sous-multigraphe (épineux) C' dans H avec $|C| = |C'|$ et $\sum_{\ell \in C} x_\ell = \sum_{(i,j,k) \in C'} X^k_{ij}$.

Démonstration.

Cette proposition découle de certaines définitions précédentes et est facile à vérifier. Soit C un trou impair dans $L(H)$ de longueur $\sum_{\ell \in C} x_\ell$. Soit e et f deux sommets de $L(H)$ correspondant respectivement aux arcs (i,j,k) et (i',j',k') dans H. Supposons que e et f sont adjacents dans C ; ceci implique que (i,j,k) et (i',j',k') sont en conflit. Selon la définition 3.4.1, (i,j,k) et (i',j',k') sont aussi adjacents. En appliquant le même raisonnement à tous les sommets de C deux à deux adjacents, nous obtiendrons un cycle conflictuel (épineux) C' de H, avec $|C| = |C'|$ et $\sum_{\ell \in C} x_\ell = \sum_{(i,j,k) \in C'} X_{ij}^k$ (en utilisant la définition 3.6.1). □

Par conséquent, pour déterminer les plans coupants dans le sous-multigraphe partiel H, il suffit d'identifier les trous impairs (cycles impairs sans corde) existant dans le graphe $L(H)$ à l'aide de la méthode des plans coupants introduite par Nemhauser et Sigismondi (Nemhauser et *al.*, 1992) (avec quelques modifications, qui seront précisées au fur et à mesure). Nous utilisons la méthode de Dijkstra pour la recherche du plus court chemin dans le graphe $B(H)$ de chaque sommet $i \in F'$ à $copie(i) \in F''$. Ce dernier correspond à un cycle impair dans le graphe $L(H)$. Seuls les cycles impairs de longueur inférieure à $1 - 2SeuilVio$, avec $SeuilVio > 0$ (où $SeuilVio$ est une borne inférieure fixée pour $\sum_{v \in C} x_v - \frac{|C|-1}{2}$) seront retenus. Nous ferons ensuite le traitement des cordes, afin de déterminer les trous impairs.

3.6.4 Traitement des cordes

Soit C un cycle impair de $L(H)$, constitué des sommets $\{v_0, v_1, ..., v_{2k}, v_{2k+1} = v_0\}$; les indices des sommets de C sont ordonnés dans le sens des aiguilles d'une montre à partir du sommet v_0 (v_0 est le sommet source utilisé pour trouver le plus court chemin).

S'il existe une ou plusieurs cordes dans le cycle impair C, alors chaque corde $\{v_d, v_f\}$ partitionne C en un cycle impair C_{df} et un chemin P avec un nombre pair de sommets et un nombre impair d'arêtes égal au nombre de sommets moins 1 (voir la

figure 3.7).

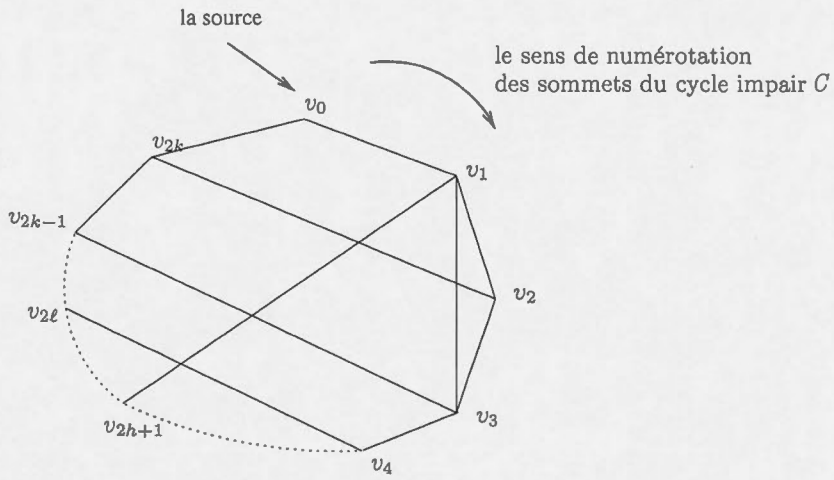

FIGURE 3.7 Un cycle impair contenant des cordes

Définition 3.6.8

Soit $\{v_{d_1}, v_{f_1}\}$ *et* $\{v_{d_2}, v_{f_2}\}$ *deux cordes dans le même cycle impair* $\{v_0, v_1, ..., v_{2k}\}$. *On dit que* $\{v_{d_2}, v_{f_2}\}$ *domine* $\{v_{d_1}, v_{f_1}\}$ *si et seulement si :*

- $d_2 \geq d_1$ *et*
- $f_2 \leq f_1$.

La figure 3.8 est une illustration de cette définition.

Proposition 3.6.9

Supposons que le chemin P *est composé des sommets* $\{v_{f+1}, v_{f+2}, ..., v_{2k}, v_0, v_1, ..., v_{d-1}\}$ *(où le nombre de sommets de* P *est égal à* $|P|$*) et qu'il existe une corde reliant les sommets* v_d *et* v_f. *Nous rappelons que le poids de chaque arête* v_iv_j *de* $L(H)$ *est* $p(v_iv_j) = 1 - x_{v_i} - x_{v_j}$ *où* x_{v_i} *(resp.* x_{v_j}*) est le poids du sommet* v_i *(resp.* v_j*). Afin de faciliter l'identification des trous impairs, nous ajoutons* ϵ *(une très petite valeur positive) au poids de chaque arête du graphe* $L(H)$. *Alors*

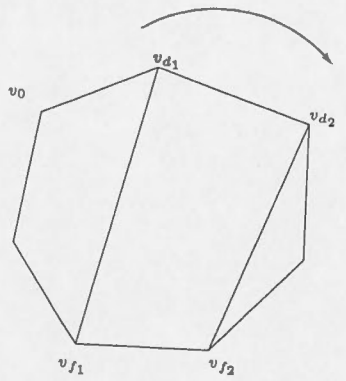

FIGURE 3.8 Exemple où $\{v_{d_2}, v_{f_2}\}$ domine $\{v_{d_1}, v_{f_1}\}$

1. *la somme des poids des arêtes formant le cycle impair C_{df} est strictement inférieure à la somme des poids des arêtes formant le cycle impair C ;*
2. *le cycle C_{df} ne peut pas contenir le sommet source v_0 ;*
3. *le cycle impair C_{df} engendré par la corde $\{v_d, v_f\}$ est un trou impair si et seulement s'il n'existe aucune corde qui domine (v_d, v_f) ;*
4. *dans un cycle impair C, il peut exister plusieurs trous impairs, qui fournissent des inégalités valides incomparables.*

Démonstration.

1. Nous désirons montrer que

   $$\sum_{v_iv_j \in C_{df}} p(v_iv_j) < \sum_{v_iv_j \in C} p(v_iv_j).$$

   Nous savons que

   $$\sum_{v_iv_j \in C} p(v_iv_j) = \sum_{v_iv_j \in C_{df}} p(v_iv_j) - (1 - x_{v_d} - x_{v_f} + \epsilon) + \sum_{v_iv_j \in C \setminus C_{df}} p(v_iv_j) \quad (3.5)$$

   $$\sum_{v_iv_j \in C \setminus C_{df}} p(v_iv_j) = \sum_{v_iv_j \in P} (1 - x_{v_i} - x_{v_j} + \epsilon) + (1 - x_{v_f} - x_{v_{f+1}} + \epsilon) + (1 - x_{v_{d-1}} - x_{v_d} + \epsilon)$$

$$\sum_{v_iv_j \in C \setminus C_{df}} p(v_iv_j) = (1 - x_{v_{f+1}} - x_{v_{f+2}} + \epsilon) + (1 - x_{v_{f+2}} - x_{v_{f+3}} + \epsilon) + ... + (1 - x_{v_{2k}} - x_{v_0} + \epsilon) + (1 - x_{v_0} - x_{v_1} + \epsilon) + ... + (1 - x_{v_{d-3}} - x_{v_{d-2}} + \epsilon) + (1 - x_{v_{d-2}} - x_{v_{d-1}} + \epsilon) + (1 - x_{v_f} - x_{v_{f+1}} + \epsilon) + (1 - x_{v_{d-1}} - x_{v_d} + \epsilon)$$

$$\sum_{v_iv_j \in C \setminus C_{df}} p(v_iv_j) = \Big(|P|+1\Big) - 2(x_{v_{f+1}} + x_{v_{f+2}} + ... + x_{v_{2k}} + x_{v_0} + x_{v_1} + ... + x_{v_{d-2}} + x_{v_{d-1}}) + \Big(|P|+1\Big)\epsilon - x_{v_f} - x_{v_d}$$

$$\sum_{v_iv_j \in C \setminus C_{df}} p(v_iv_j) = |P| - 2\sum_{v_k \in P} x_{v_k} + |P|\epsilon + \Big(1 - x_{v_d} - x_{v_f} + \epsilon\Big)$$

$$\sum_{v_iv_j \in C \setminus C_{df}} p(v_iv_j) = (1 - x_{v_d} - x_{v_f} + \epsilon) + 2\Big(\frac{|P|}{2} - \sum_{v_k \in P} x_{v_k} + \frac{|P|}{2}\epsilon\Big)$$

Nous avons déjà montré que $\frac{|P|}{2} - \sum_{v_k \in P} x_{v_k} \geq 0$ (voir 3.4 page 32) ; nous savons aussi que $\epsilon > 0$ et $|P| > 1$. D'où

$$\frac{|P|}{2} - \sum_{v_k \in P} x_{v_k} + \frac{|P|}{2}\epsilon > 0. \tag{3.6}$$

D'où

$$\sum_{v_iv_j \in C \setminus C_{df}} p(v_iv_j) - (1 - x_{v_d} - x_{v_f} + \epsilon) > 0. \tag{3.7}$$

Les équations 3.5 et 3.7 nous permettent de conclure que :

$$\sum_{v_iv_j \in C} p(v_iv_j) > \sum_{v_iv_j \in C_{df}} p(v_iv_j) \tag{3.8}$$

2. Supposons que le cycle impair C_{df} contient la source v_0. Ceci implique qu'il existe un plus court chemin de v_0 à $copie(v_0)$ dans le graphe biparti $B(H)$, de la même longueur que le cycle C_{df}. Comme la longueur de C_{df} est inférieure à celle de C (voir le point précédent), le chemin qui a fourni le cycle C n'est pas le plus court chemin, ce qui est une contradiction.

3. La démonstration de la troisième observation est triviale, puisque par définition un trou est un cycle impair sans corde avec au moins 5 sommets.

4. Dans un même cycle impair, il peut exister plusieurs cordes où aucune corde ne domine l'autre (voir la figure 3.7). Chaque corde nous permet d'identifier un trou impair qui possède au moins deux arêtes qui n'appartiennent à aucun autre trou impair. Par conséquent, chaque trou impair fournit une inégalité valide différente des autres inégalités valides correspondant aux autres trous impairs. □

Les propriétés précédentes nous permettent d'écrire l'algorithme 3.3. Ce dernier effectue le traitement des cordes dans un cycle impair donné.

Exemple

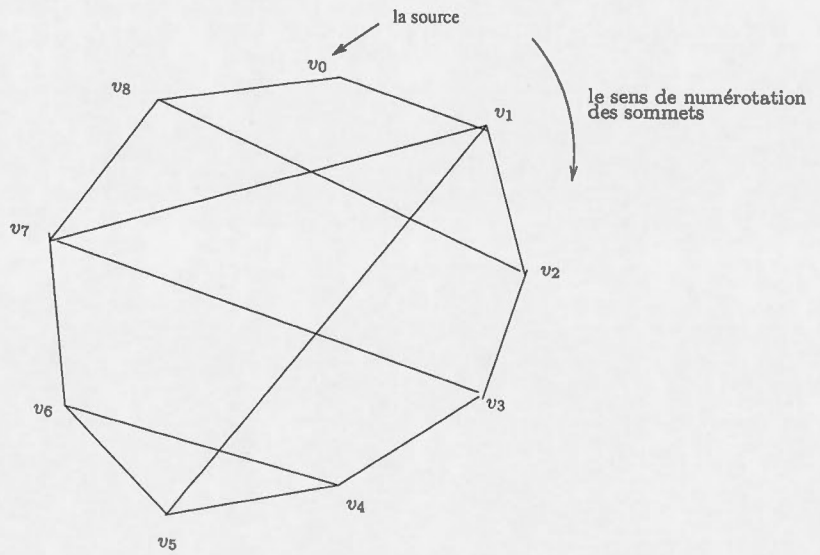

FIGURE 3.9 Exemple de cordes

Dans l'exemple de la figure 3.9, nous avons un cycle impair C, constitué des sommets $\{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$, et il existe cinq cordes $\{v_1, v_5\}$, $\{v_1, v_7\}$, $\{v_2, v_8\}$, $\{v_3, v_7\}$ et $\{v_4, v_6\}$. En appliquant la définition 3.6.8, nous constatons que les cordes $\{v_1, v_5\}$, $\{v_3, v_7\}$ et $\{v_4, v_6\}$ dominent la corde $\{v_1, v_7\}$, $\{v_3, v_7\}$ et $\{v_4, v_6\}$ dominent la corde $\{v_2, v_8\}$ et $\{v_4, v_6\}$ domine $\{v_3, v_7\}$. Par conséquent, seules les cordes $\{v_1, v_5\}$ et

$\{v_4, v_6\}$ ne sont dominées par aucune autre corde. Le cycle C_{15} composé des sommets $\{v_1, v_2, v_3, v_4, v_5\}$ est donc un trou impair, de même que le triangle C_{46} composé des sommets $\{v_4, v_5, v_6\}$.

3.6.5 Procédure de liftage (procédure de A. Hadjar)

Soit $C' = \{a_0 = a_{2k+1}, a_1, a_2, ..., a_{2k}\}$ un cycle conflictuel (épineux). Rappelons que pour $i = 0, 1, ..., 2k$, les arcs a_i et a_{i+1} sont en conflit.

Définition 3.6.10

Un ensemble Q d'arcs du sous-multigraphe partiel H est une clique de conflit si tous les arcs de Q sont deux à deux en conflit.

À partir des arcs du cycle conflictuel (épineux) C', nous construisons un ensemble $\mathbf{Q}$ des cliques $\{Q_1, Q_2, ..., Q_{2k+1}\}$, où $Q_i = \{a_{i-1}, a_i\}$, pour $i = 1, 2, ..., 2k+1$. Initialement chaque clique de conflit contient deux arcs qui sont adjacents et en conflit.

Comme deux arcs en conflit ont au moins une extrémité en commun, désignons par $S(Q_i)$ l'ensemble des extrémités communes à tous les arcs de Q_i.

$$S(Q_i) = \begin{cases} \{s\} & \text{si } a_{i-1} \text{ et } a_i \text{ ne sont pas des arcs parallèles et} \\ & s \text{ est l'extrémité commune de } a_{i-1} \text{ et } a_i \\ \{u, v\} & \text{si } a_{i-1} \text{ et } a_i \text{ sont des arcs parallèles où } u \\ & \text{et } v \text{ sont leurs extrémités communes} \end{cases}$$

Notons que deux cliques distinctes peuvent avoir le même ensemble d'extrémités.

Soit F l'ensemble des arcs de H. Etant donné un arc $a = (u, v, k) \in F$, soit $K(a) = \{Q \in \mathbf{Q} : S(Q) \cap \{u, v\} \neq \emptyset\}$.

L'algorithme 3.4 effectue le liftage avec des arcs de flux non nul (strictement positifs). Nous aurons besoin par la suite de faire un liftage avec des arcs de flux nul;

Algorithme 3.3 TraitementCordes($C, ListCyc$)

1: Soit $C' = \{a_0, a_1, ..., a_{2k}\}$ le cycle conflictuel de H correspondant au cycle impair $C = \{v_0, v_1, ..., v_{2k}\}$ de $L(H)$.
2: Soit la liste des cordes $ListeCordes = \emptyset$
3: Soit la liste des cycles impairs $ListeCycles = \emptyset$
4: **pour** $i := 1$ **à** $2k - 1$ **faire**
5: **si** *i est pair* **alors**
6: $\ell := 2k$
7: **sinon**
8: $\ell := 2k - 1$
9: **finsi**
10: **pour** $j = i + 2$ **à** ℓ **pas** 2 **faire**
11: **si** les arcs a_i et a_j sont en conflit **alors**
12: ajouter la corde (v_i, v_j) à la liste des cordes *ListeCordes*
13: **finsi**
14: **finpour**
15: **finpour**
16: **si** *ListeCordes est vide* **alors**
17: ajouter C' à la liste des trous impairs *ListeCycles*
18: **sinon**
19: **pour** *chaque* $CORDE \in ListCorde$ **faire**
20: **si** *CORDE n'est dominée par aucune autre corde de ListeCordes* **alors**
21: ajouter le trou impair engendré par $CORDE$ à *ListeCycles*
22: **finsi**
23: **finpour**
24: **finsi**

Algorithme 3.4 LiftPos(C', F)

1: {F est l'ensemble des arcs du graphe H et à l'entrée C' est le trou impair}
2: Former les cliques Q_i, $i = 1, 2, ..., 2k + 1$ et définir pour chaque clique l'ensemble $S(Q_i)$
3: **pour** chaque arc $a = (u, v, k) \in F$ **faire**
4: $K(a) := \emptyset$
5: **si** $X^k_{uv} > 0$ et ni l'origine ni la destination de a ne sont des dépôts **alors**
6: **pour** chaque $Q_i \in$ **Q faire**
7: **si** $S(Q_i) \cap \{u, v\} \neq \emptyset$ **alors**
8: ajouter Q_i à $K(a)$
9: **finsi**
10: **finpour**
11: **si** il existe deux cliques distinctes Q_i et $Q_j \in K(a)$ **telles que** $Q_i \cup \{a\}$ **et** $Q_j \cup \{a\}$ sont des cliques **alors**
12: $Q_i := Q_i \cup \{a\}$
13: $Q_j := Q_j \cup \{a\}$
14: $C' := C' \cup \{a\}$
15: **finsi**
16: **finsi**
17: **finpour**

l'algorithme précédent reste valable et le seul changement se fera au niveau de la ligne 5, où nous remplacerons la condition $X_{uv}^k > 0$ par $X_{uv}^k = 0$. Le nouvel algorithme sera nommé LiftNul(C', F).

Exemple

Nous reprenons l'exemple de la figure 3.6 et nous appliquons la procédure de liftage. Les arcs a_i sont les arcs du cycle conflictuel épineux et les arcs b_j sont les arcs ajoutés par la procédure de liftage.

Étape initiale

$C' = \{a_0 = a_{15}, a_1, ..., a_{14}\}$ est le cycle conflictuel épineux. L'ensemble des cliques est $\mathbf{Q} = \{Q_1, Q_2, ..., Q_{15}\}$, où

$Q_1 = \{a_0, a_1\}$ et $S(Q_1) = 6$,

$Q_2 = \{a_1, a_2\}$ et $S(Q_2) = 1$,

$Q_3 = \{a_2, a_3\}$ et $S(Q_3) = 1, 2$,

...

$Q_{15} = \{a_{14}, a_0\}$ et $S(Q_{15}) = 6$.

Étape intermédiaire

Soit b_1 un arc de F ($b_1 \notin C'$) et ni son origine (le sommet 6) ni sa destination (le sommet 20) ne sont des dépôts. On remarque que $S(Q_1) \cap \{6, 20\} = \{6\}$ et $S(Q_{15}) \cap \{6, 20\} = \{6\}$, donc $K(b_1) = \{Q_1, Q_{15}\}$.

Puisque $Q_1 \cup \{b_1\}$ et $Q_{15} \cup \{b_1\}$ sont des cliques de conflit, alors $Q_1 = \{a_0, a_1, b_1\}$, $Q_{15} = \{a_{14}, a_0, b_1\}$ et $C' = \{a_0 = a_{15}, a_1, ..., a_{14}, b_1\}$

Cette étape s'applique pour chaque arc de F possédant au moins une extémité en commun avec au moins deux cliques distinctes.

Étape finale

La procédure de liftage se termine quand tous les arcs de F sont examinés. Pour cet exemple, à la fin de la procédure les cliques modifiées sont

$Q_1 = \{a_0, a_1, b_1, b_2\}$,

$Q_{15} = \{a_{14}, a_0, b_1, b_2\}$,

$Q_2 = \{a_1, a_2, b_3\}$,

$Q_5 = \{a_4, a_5, b_3\}$,

$Q_6 = \{a_5, a_6, b_4\}$,

$Q_{10} = \{a_9, a_{10}, b_4\}$,

et le cycle conflictuel épineux représenté par la figure 3.10 devient $\{a_0 = a_{15}, a_1, ..., a_{14}, b_1, b_2, b_3, b_4\}$

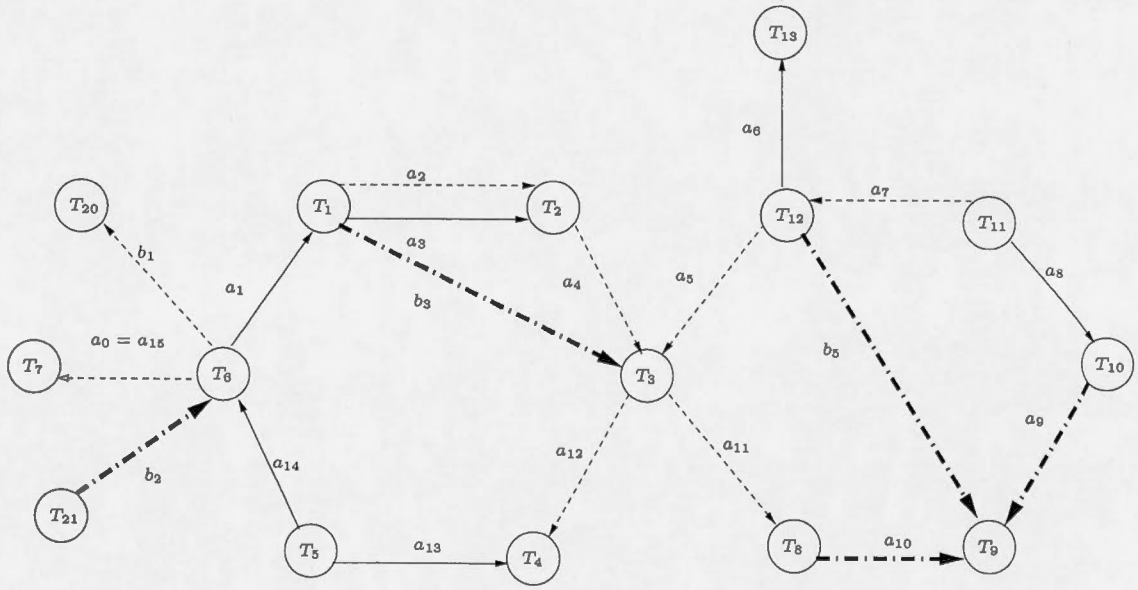

FIGURE 3.10 Exemple de liftage d'un cycle conflictuel épineux avec un point d'articulation

3.6.6 Les inégalités valides

Proposition 3.6.11

Soit H un sous-multigraphe partiel . Soit C' un cycle conflictuel (épineux) de H.

L'inégalité $\sum_{(i,j,k)\in C'} X_{ij}^k \leq \left\lfloor \frac{|C'|}{2} \right\rfloor$ *est valide pour le problème* (P) *(défini dans la section 3.2).*

Démonstration.

Avant d'effectuer le liftage, le cycle conflictuel (épineux) C' de H contient des arcs deux à deux en conflit. Au niveau de chaque sommet s du cycle, l'inégalité $\sum_{(s,j,k)\in C'} X_{sj}^k + \sum_{(i,s,k)\in C'} X_{is}^k \leq 1$ est valide pour (P). La somme de toutes les inégalités valides obtenues au niveau de chaque sommet du cycle conflictuel (épineux) C' est :

$$\sum_{s=1}^{|C'|} \left(\sum_{(s,j,k)\in C'} X_{sj}^k + \sum_{(i,s,k)\in C'} X_{is}^k \right) \leq \sum_{s=1}^{|C'|} 1 = |C'| \tag{3.9}$$

Comme chaque arc possède deux extrémités, chaque X_{ij}^k apparaît deux fois dans la somme précédente, d'où :

$$2 \sum_{(i,j,k)\in C'} X_{ij}^k \leq |C'| \tag{3.10}$$

$$\iff \sum_{(i,j,k)\in C'} X_{ij}^k \leq \left\lfloor \frac{|C'|}{2} \right\rfloor. \tag{3.11}$$

□

Remarque 3.6.12

Selon la définition 3.4.2, le cycle conflictuel (épineux) C' est un sous-multigraphe partiel conflictuel (épineux) de G. Le cycle impair C'' obtenu en liftant le cycle conflictuel (épineux) C' est aussi un sous-multigraphe partiel conflictuel de G.

Proposition 3.6.13

Soit C'' le cycle conflictuel obtenu en liftant le cycle conflictuel (épineux) C' de H, où $H = (S, F)$ est le sous-multigraphe partiel représentant le support de la solution non-entière obtenue. L'inégalité $\sum_{(i,j,k)\in C''} X_{ij}^k \leq \left\lfloor \frac{|C'|}{2} \right\rfloor$ *est valide pour le problème (P) (défini dans la section 3.2).*

Démonstration.

En liftant un cycle conflictuel (épineux) C', nous gardons les mêmes sommets et chaque arc ajouté est en conflit avec tous les arcs qui lui sont adjacents, nous avons donc $|C'|$ sommets dans le cycles C''. Et au niveau de chaque sommet s de C'', $\sum_{(s,j,k)\in C''} X_{sj}^k + \sum_{(i,s,k)\in C''} X_{is}^k \leq 1$ est valide pour (P). La somme de toutes les inégalités valides obtenues au niveau de chaque sommet du cycle conflictuel (épineux) C'' est :

$$\sum_{s=1}^{|C''|} \left(\sum_{(s,j,k)\in C''} X_{sj}^k + \sum_{(i,s,k)\in C''} X_{is}^k \right) \leq |C'| \tag{3.12}$$

Comme chaque arc possède deux extrémités, X_{ij}^k apparaît deux fois dans la somme précédente, d'où :

$$2 \sum_{(i,j,k)\in C''} X_{ij}^k \leq |C'| \tag{3.13}$$

$$\iff \sum_{(i,j,k)\in C''} X_{ij}^k \leq \left\lfloor \frac{|C'|}{2} \right\rfloor . \tag{3.14}$$

□

Remarque 3.6.14

Une violation est la différence entre le premier et le second membre de l'inégalité valide. Si nous désirons éviter les petites violations, nous pouvons fixer un seuil de violation, noté $SeuilVio$, au départ.

Il est possible de chercher toutes les inégalités valides, ensuite de vérifier si la valeur de la violation est supérieure à $SeuilVio$; dans ce cas, l'inégalité valide est ajoutée au modèle. Mais nous pouvons améliorer le temps d'exécution en supprimant certains sommets et arcs du graphe biparti $B(H)$ et certains plus courts chemins avant même de faire le traitement des cordes. Nous avons remarqué qu'il existe une relation entre le poids du plus court chemin et le seuil de violation, et par conséquent, entre ce dernier et le poids des arcs de $B(H)$.

Lemme 3.6.15

Soit C' un cycle conflictuel (épineux) du graphe H et $SeuilVio$ un seuil de violation fixé. La relation $\sum_{(i,j,k)\in C'} X_{ij}^k - \left\lfloor \frac{|C'|}{2} \right\rfloor \geq SeuilVio$ est vérifiée si et seulement si le poids du plus court chemin de v_0 à v_0' dans $B(H)$ est inférieur ou égal à $1 - 2SeuilVio$.

Démonstration.

Nous désirons que la violation soit supérieure ou égale à $SeuilVio$. Nous avons donc

$$\sum_{(i,j,k)\in C'} X_{ij}^k - \left\lfloor \frac{|C'|}{2} \right\rfloor \geq SeuilVio \tag{3.15}$$

$$\iff 2 \sum_{(i,j,k)\in C'} X_{ij}^k - (|C'| - 1) \geq 2SeuilVio \tag{3.16}$$

$$\iff 2 \sum_{(i,j,k)\in C'} X_{ij}^k - |C'| \geq 2SeuilVio - 1 \tag{3.17}$$

$$\Longleftrightarrow |C'| - 2 \sum_{(i,j,k)\in C'} X_{ij}^k \leq 1 - 2SeuilVio. \tag{3.18}$$

Nous avons déjà montré qu'à tout trou impair C de $L(H)$ correspond un cycle conflictuel (épineux) C' de H, tel que $|C| = |C'| = 2k+1$ et $\sum_{v_\ell \in C} x_{v_\ell} = \sum_{(i,j,k)\in C'} X_{ij}^k$ (voir la proposition 3.6.7). Par conséquent la relation 3.18 devient

$$|C| - 2 \sum_{v_\ell \in C} x_{v_\ell} \leq 1 - 2SeuilVio \Longleftrightarrow \tag{3.19}$$

$$2k + 1 - 2 \sum_{v_\ell \in C} x_{v_\ell} \leq 1 - 2SeuilVio. \tag{3.20}$$

L'expression $\left(2k + 1 - 2 \sum_{v_\ell \in C} x_{v_\ell}\right)$ représente le poids du plus court chemin de v_0 à v_0' dans $B(H)$ et l'expression est aussi égale à la somme des poids des arêtes du trou impair C dans $L(H)$ (voir l'équation 3.1 dans la section 3.5). □

Lemme 3.6.16

Si le poids du plus court chemin de v_0 à v_0' dans $B(H)$ est inférieur ou égal à $1 - 2SeuilVio$, alors le poids de chaque arc du plus court chemin de v_0 à v_0' est inférieur ou égal à $1 - 2SeuilVio$.

Démonstration.

Le poids du trou impair est égal à la somme des poids des arêtes appartenant à ce trou. Puisque le poids du trou impair est inférieur à $1 - 2SeuilVio$ (ce qui est prouvé dans le lemme précédent), alors le poids de chaque arête est aussi inférieur à $1 - 2SeuilVio$ (car le poids de chaque arête uv dans $L(H)$ est non négatif puisqu'il est égal à $1 - x_u - x_v \geq 0$). □

Par conséquent, tous les arcs dont le poids est supérieur à $1 - 2SeuilVio$ sont inutiles ; car si au moins un arc de poids supérieur à $1 - 2SeuilVio$ appartient au plus court chemin de v_0 à v_0' dans $B(H)$, alors le poids de ce chemin sera supérieur à

$1 - 2SeuilVio$. Dans ce cas, il est préférable de les éliminer ; ceci nous permettra de diminuer la dimension du graphe $L(H)$, et par conséquent, celle du graphe $B(H)$. Nous allons donc redéfinir le graphe adjoint $L(H)$.

Définition 3.6.17

Soit $L(H) = (F, L)$ le graphe adjoint (non orienté) associé au sous-multigraphe partiel $H = (S, F)$. S'il existe deux arcs (i, j, k) et (i', j', k') de H qui sont en conflit et si $1 - X_{ij}^{k} - X_{i'j'}^{k'} \leq 1 - 2SeuilVio$, alors,

1. *pour chaque arc nous créons un sommet dans $L(H)$, s'il n'existe pas encore,*
2. *x_u est le poids du sommet u, c'est-à-dire le poids de l'arc que le sommet u représente,*
3. *les deux sommets représentant les arcs (i, j, k) et (i', j', k') seront reliés par une arête (s'ils sont en conflit), et*
4. *le poids de l'arête $uv \in L$ est égal à $1 - x_u - x_v$.*

3.7 La séparation des inégalités valides du MDVSP

La méthode de séparation des inégalités valides du MDVSP que nous proposons est simple. Elle commence par l'extraction de la solution de la relaxation linéaire ; à partir de cette dernière, les graphes $H = (S, F)$, $L(H) = (F, L)$ et $B(H) = (F', F'', L')$ seront créés. Ensuite, pour chaque sommet de F', nous chercherons le plus court chemin, afin d'identifier les trous impairs, s'ils existent. Nous enrichirons chaque trou impair trouvé avec un liftage par des arcs de flux non nul ; ensuite le cycle impair obtenu sera lifté par des arcs de flux nul. Avant la procédure de liftage par des arcs de flux non nul et après, nous devons nous assurer que chaque trou impair n'est utilisé qu'une seule fois. Rappelons qu'un cycle impair lifté (ainsi que le trou impair) correspond à une inégalité valide. Après avoir trouvé les inégalités valides, elles seront ajoutées au problème à résoudre. La méthode est décrite dans l'algorithme 3.5.

Algorithme 3.5 SéparationInégValid($Solution, ListInégValid$)

1: extraire la solution de la relaxation linéaire
2: fixer le paramètre $SeuilVio$
3: créer les graphes $H = (S, F)$, $L(H) = (F, L)$ et $B(H) = (F', F'', L')$
4: **pour** chaque sommet $i \in F'$ **faire**
5: Dijkstra($B(H), w, i$)
6: **si** il existe un plus court chemin de i à $copie(i)$ de poids inférieur à $1 - 2SeuilVio$ **alors**
7: C := le cycle impair dans $L(H)$ correspondant au plus court chemin de i à $copie(i)$ dans $B(H)$
8: TraitementCordes($C, ListCyc$)
9: **finsi**
10: **finpour**
11: **pour** chaque cycle $C' \in ListCyc$ **faire**
12: LiftPos(C',F)
13: **si** C' n'existe pas dans $ListCyc$ (après liftage) **alors**
14: ajouter C' à $ListCyc$
15: **finsi**
16: **finpour**
17: trier la liste des cycles $ListCyc$ par ordre décroissant selon la violation (dans le cas où le nombre d'inégalités valides est limité cette étape nous permet de choisir celles qui possédent les plus grandes violations)
18: **pour** chaque cycle $C' \in ListCyc$ **faire**
19: LiftNul(C',F)
20: ajouter l'inégalité valide relative à C' à $ListInégValid$
21: **finpour**

3.8 Algorithme de résolution du MDVSP

Pour la résolution du MDVSP, nous proposons un algorithme de type « Branch-and-Cut », qui fonctionne selon le principe présenté dans le chapitre II à la page 18, mais les plans coupants (étape 6) sont générés avec la méthode de séparation des inégalités valides présentée dans la section précédente.

Au noeud racine de l'arbre de branchement, l'algorithme commence par résoudre la relaxation linéaire du problème (P) présentée au début du chapitre III. S'il existe une solution optimale et si elle n'est pas entière, alors l'algorithme cherche des plans coupants à l'aide de la méthode de séparation représentée par l'algorithme 3.5. S'il existe des plans coupants, l'algorithme ajoute les inégalités valides au programme linéaire et le résout de nouveau (au niveau du même noeud). La même opération est répétée jusqu'à qu'il n'existe plus de coupes ; dans ce cas, deux noeuds fils sont créés. Dans l'un, une variable X_{ij}^k est fixée à 0, et dans l'autre, la même variable est fixée à 1. Le choix de la variable de branchement est fait par le logiciel utilisé pour la résolution du problème. Au niveau de chaque noeud de l'arbre de branchement, l'algorithme procède de la même façon pour résoudre la relaxation du sous-problème. Si la solution optimale existe et n'est pas entière, alors la méthode de séparation est appliquée jusqu'à ce qu'il n'existe plus de plans coupants ; dans ce cas, deux noeuds fils sont créés et ainsi de suite jusqu'à ce qu'il n'existe plus de noeuds actifs dans l'arbre de branchement.

Nous avons fait plusieurs expériences et comparé les résultats obtenus par l'ajout de toutes les coupes détectées et les résultats obtenus par la méthode de « Branch-and-Cut » utilisée par CPLEX. Nous avons constaté que le nombre de noeuds de l'arbre de branchement de notre algorithme est inférieur au nombre de noeuds de l'arbre de branchement de CPLEX ; par contre, son temps de résolution est supérieur à celui de CPLEX. Ceci est dû au fait que les coupes générées sont très nombreuses. Pour cette raison nous avons décidé de limiter le nombre de coupes ajoutées ; le chapitre suivant contient plus de détails sur ce point.

CHAPITRE IV

TESTS ET RÉSULTATS

4.1 Introduction

Ce dernier chapitre est consacré à l'implémentation, la description des données et des scénarios testés par notre algorithme, et la présentation des résultats des tests et comparaisons. Nous effectuons une première comparaison de sept scénarios ; ensuite, nous choisissons les trois meilleurs et nous les comparons avec la méthode de « Branch-and-Cut » de CPLEX. Afin de tester l'efficacité des coupes générées par notre algorithme, nous avons effectué une autre comparaison avec un algorithme produisant plus de coupes. Nous terminons par une comparaison d'un de nos scénarios avec la méthode de séparation et évaluation progressive.

4.2 Implémentation

Notre algorithme a été implémenté en utilisant le langage C et une structure de tas (heap) de C++ sous l'environnement UNIX. Pour la résolution du programme linéaire en nombres entiers, nous avons fait appel au logiciel CPLEX (version 10.0) développé par ILOG. Pour l'ajout des coupes locales, c'est-à-dire à un noeud donné de l'arbre de branchement, lorsqu'il en existe, nous utilisons la fonction « CPXcutcallbackaddlocal ». Cette dernière ajoute les coupes au programme linéaire du sous-problème associé au noeud courant. Les coupes ajoutées sont valables pour le sous-arbre engendré par le noeud courant.

La procédure cutCyclecallback détermine nos coupes et fait appel à la fonction « CPXcutcallbackaddlocal », qui effectue l'ajout des coupes au noeud courant. Pour que ILOG CPLEX appelle « cutCyclecallback » à chaque noeud de l'arbre de branchement, nous avons besoin de faire appel à la fonction « CPXsetcallbackfunc », avant l'appel de CPXmipopt. Mais avant tout, nous devons fixer certains paramètres et cela en utilisant la fonction « CPXsetintparam ». Le détail des fonctions et paramètres est donné en annexe.

Dans le but d'optimiser le temps d'exécution du programme et surtout l'espace mémoire utilisé, nous n'avons pas créé le sous-multigraphe partiel H et le graphe adjoint correspondant $L(H)$, mais directement le graphe biparti $B(H)$.

Notons que pour certains tests, nous avons modifié les paramètres d'ajout des coupes de CPLEX (qui peuvent varier entre -1 à 3) ; pour les comparaisons de la section 4.5 et de la section 4.6, nous avons gardé les paramètres par défaut à 0, c'est-à-dire que nous laissons CPLEX choisir quand faire la recherche des coupes. Dans les comparaisons de la section 4.7, nous avons changé les paramètres de sorte que lorsque nous faisons des tests en utilisant notre scénario, les paramètres sont tous fixés à -1, c'est-à-dire que nous interdisons à CPLEX d'ajouter ses propres coupes ; donc seulement les coupes générées par notre algorithme sont ajoutées aux sous-problèmes. Par contre, en testant la méthode « Branch-and-Cut » appliquée par CPLEX, nous fixons les paramètres à 1, c'est-à-dire que nous demandons à CPLEX de chercher un nombre modéré de coupes qui peuvent être efficaces.

En plus du critère d'arrêt du branchement de la méthode « Branch-and-Cut », CPLEX utilise deux autres paramètres, le « *GAP absolu* » et le « *GAP relatif* ». Le premier est la différence entre l'objectif de la meilleure solution entière et le meilleur objectif des problèmes relaxés (en pratique, le meilleur objectif est celui du noeud racine). Et le *GAP relatif* est égal à $\frac{|GAP\ absolu|}{\epsilon+|objectif\ de\ la\ meilleure\ solution\ entière|}$. Lorsque chaque paramètre est inférieur à un seuil de tolérance donné, CPLEX arrête le branchement. Nous avons modifié les seuils de tolérance que CPLEX utilise par défaut afin de s'as-

surer que la solution trouvée, si elle n'est pas optimale, est la meilleure possible. Nous avons fixé les deux seuils de tolérance à 10^{-6} ; c'est-à-dire que tant que les valeurs du « *GAP absolu* » et le du « *GAP relatif* » sont supérieures à 10^{-6}, CPLEX continue la résolution du problème.

4.3 Description des données

Afin de tester notre algorithme, nous avons utilisé un générateur aléatoire des instances du MDVSP qui a été crée par Carpaneto et *al.* (1989). Ce générateur a été utilisé par plusieurs chercheurs (Ribeiro et Soumis (1994), Fischetti et *al.*(2001), Hadjar, Marcotte et Soumis (2006) ainsi que d'autres) pour tester leur algorithme. Les résultats présentés dans nos tableaux sont les moyennes de 10 instances générées de type A ou B (pour plus de détails voir (Carpaneto et *al.*, 1989)), pour un nombre donné de tâches et de dépôts. Par exemple, pour un problème de 300 tâches avec 4 dépôts, le nombre de noeuds est égal à la moyenne des nombres de noeuds de 10 instances différentes de 300 tâches avec 4 dépôts.

4.4 Scénarios testés

Nous avons testé plusieurs scénarios, mais nous présenterons seulement les plus importants ou les plus significatifs. Dans un premier temps, nous avons ajouté toutes les inégalités valides trouvées, mais le temps de résolution était très important ainsi que le nombre de coupes générées ; par exemple, pour une instance de 450 tâches de type A avec 6 dépôts, l'algorithme a généré 10187 coupes, parmi lesquelles 7390 coupes ont une violation inférieure à 0.1, 2117 coupes ont une violation qui varie entre 0.1 et 0.2, 573 coupes ont une violation entre 0.2 et 0.3, 107 coupes ont une violation comprise entre 0.3 et 0.4, et finalement 28 coupes ont une violation entre 0.4 et 0.5. Nous avons donc décidé de limiter le nombre d'inégalités valides ajoutées. Rappelons que pour cette première comparaison, nous avons permis l'ajout des coupes générées par CPLEX en plus de celles générées par notre méthode.

Avant de présenter nos scénarios, rappelons que $SeuilVio$ dénote la borne inférieure du seuil de violation. Précisons aussi que la procédure de séparation des inégalités valides mentionnée est celle décrite par l'algorithme 3.5.

- **Scénario 1**
  1. $SeuilVio = 0.03$
  2. Si le nombre de coupes ajoutées est au moins égal à $Max\{10, n/10\}$, alors cesser de faire appel à la procédure de séparation des inégalités valides.
  3. Si un sommet de $B(H)$ appartient à un trou impair détecté, alors ne pas le prendre comme source pour la recherche du plus court chemin.
- **Scénario 2**
  1. $SeuilVio = 0.02$
  2. Si le nombre de coupes ajoutées est au moins égal à $Max\{10, n/10\}$, alors cesser de faire appel à la procédure de séparation des inégalités valides.
  3. Si un sommet de $B(H)$ appartient à un trou impair détecté, alors ne pas le prendre comme source pour la recherche du plus court chemin.
- **Scénario 3**
  1. $SeuilVio = 0.01$
  2. Si le nombre de coupes ajoutées est égal à $Max\{10, n/10\}$, alors cesser de faire appel à la procédure de séparation des inégalités valides.
  3. Si un sommet de $B(H)$ appartient à un trou impair détecté, alors ne pas le prendre comme source pour la recherche du plus court chemin.
- **Scénario 4**
  1. $SeuilVio = 0.01$
  2. Si le nombre de coupes ajoutées est égal à $Max\{10, n/10\}$, alors cesser de faire appel à la procédure de séparation des inégalités valides.
- **Scénario 5**
  1. $SeuilVio = 0.01$

2. Si le nombre de coupes ajoutées est égal à $Max\{10, n/20\}$, alors cesser de faire appel à la procédure de séparation des inégalités valides.

- **Scénario 6**

1. $SeuilVio = 0.01$
2. Si le nombre de coupes ajoutées est au moins égal à $Max\{10, n/20\}$, alors cesser de faire appel à la procédure de séparation des inégalités valides.
3. Si un sommet de $B(H)$ appartient à un trou impair détecté, alors ne pas le prendre comme source pour la recherche du plus court chemin.

- **Scénario 7**

1. $SeuilVio = 0.02$
2. Si le nombre de coupes ajoutées est égal à $Max\{5, n/20\}$, alors cesser de faire appel à la procédure de séparation des inégalités valides.
3. Si un sommet de $B(H)$ appartient à un trou impair détecté, alors ne pas le prendre comme source pour la recherche du plus court chemin.

4.5 Comparaison des scénarios

Afin de choisir un seul scénario qui soit le meilleur de tous les scénarios présentés dans la section pécédente, nous avons effectué une première comparaison avec des problèmes de type A et B de taille variable et avec 4 dépôts. Les résultats sont présentés dans les tableaux 4.1 et 4.2. Chaque tableau contient les informations suivantes.

- Nbr Tâches : représente le nombre de tâches du problème
- Numéro Scénario : les scénarios sont numérotés de 1 à 7
- Nbr Noeuds : le nombre de noeuds de l'arbre de branchement
- Nbr Coupes : le nombre total de coupes ajoutées au problème
- Temps $B(H)$: le temps de construction du graphe biparti $B(H)$ en secondes
- Temps $CycImp$: le temps de génération des cycles impairs en secondes
- Temps de liftage : le temps total de liftage des cycles impairs en secondes

- Temps Total CPU : le temps CPU de résolution du problème (égal au Temps $B(H)$ + Temps $CycImp$ + Temps de liftage + temps de branchement + temps de résolution des sous-problèmes au niveau de chaque noeud de l'arbre)

En analysant les résultats des tableaux 4.1 et 4.2 pour la comparaison des scénarios, nous remarquons en premier qu'il n'existe pas de scénario meilleur que tous les autres. Pour les problèmes de type B, le scénario 7 donne le meilleur temps CPU de résolution pour les problèmes de 350, 400 et 450 tâches ; par contre, pour les problèmes de 500 et 550 tâches c'est le scénario 2 qui fournit le meilleur temps de résolution, bien que les temps de construction de $B(H)$, de génération des cycles impairs et de liftage soient supérieurs à ceux du scénario 7 et même que le nombre de coupes ajoutées soit le double. Nous ne faisons pas les mêmes constatations en comparant les problèmes de type A. Le meilleur temps de résolution des problèmes de 350 et 500 tâches est enregistré par le scénario 2 ; les scénarios 5 et 7 résolvent plus rapidement les problèmes de 400 et 450 tâches, respectivement. Les résultats montrent aussi que les temps de construction du graphe biparti $B(H)$, de recherche des cycles impairs et de liftage sont très modestes par rapport au temps total de résolution.

4.6 Comparaison de l'algorithme de « Branch-and-Cut » de CPLEX et de certains scénarios

En nous basant sur les résultats de la section précédente, nous avons décidé d'effectuer d'autres comparaisons entre les scénarios 2, 5 et 7 et la méthode « Branch-and-Cut » appliquée par le logiciel CPLEX. Notons que, en résolvant les problèmes à l'aide du « Branch-and-Cut » de CPLEX, nous n'avons pas changé les paramètres d'ajout des coupes du logiciel.

Pour comparer les nombres de noeuds des arbres de branchement et les temps totaux de résolution des problèmes, nous calculons pour chaque scénario la différence entre le nombre de noeuds (resp. temps de résolution) du scénario et celui de la méthode « Branch-and-Cut » de CPLEX par rapport à cette dernière (en pourcentage). Nous avons effectué des tests avec des problèmes de type A et B de 100, 150, 200, 250, 300,

TABLEAU 4.1 Comparaison des scénarios pour les problèmes de type A avec 4 dépôts

Nbr Tâches	Numéro Scénario	Nbr Noeuds	Nbr Coupes	Temps $B(H)(s)$	Temps CycImp(s)	Temps de liftage(s)	Temps Total CPU(s)
350	(1)	34.9	30.6	1.128	4.807	2.594	56.61
	(2)	37.8	30.6	1.114	4.705	2.531	**53.346**
	(3)	37.8	30.6	1.168	5.02	2.658	57.779
	(4)	42.3	30.6	1.052	4.578	2.601	54.969
	(5)	41.3	15.3	0.692	3.137	1.512	57.729
	(6)	40.3	15.3	0.717	3.027	1.51	61.262
	(7)	40.3	15.3	0.656	2.762	1.429	58.097
400	(1)	102.9	40	1.984	8.901	4.808	125.463
	(2)	101.3	40	1.941	8.852	4.689	128.588
	(3)	110.6	40	2.013	9.336	4.88	132.782
	(4)	117.3	40	1.955	9.183	4.934	131.448
	(5)	153.3	20	1.23	5.807	2.681	**121.5**
	(6)	165.8	20	1.113	4.966	2.534	122.352
	(7)	167.8	20	1.138	5.082	2.49	123.578
450	(1)	116.9	39.6	2.128	8.9	5.726	208.449
	(2)	139.3	39.6	1.872	8.109	5.454	197.899
	(3)	118.1	39.6	1.959	8.583	5.607	216.775
	(4)	108.1	39.6	1.877	8.613	5.87	200.998
	(5)	143.1	19.8	1.409	6.379	3.493	180.845
	(6)	160	19.8	1.409	6.161	3.477	172.388
	(7)	145.9	19.8	1.232	5.245	3.169	**149.591**
500	(1)	440.5	49.6	3.43	14.87	9.822	328.213
	(2)	405.6	49.6	2.842	12.281	9.285	**284.176**
	(3)	421.4	49.6	2.822	12.545	9.828	368.997
	(4)	407.8	49.8	2.501	11.454	9.573	310.422
	(5)	660	24	2.159	10.02	5.358	300.283
	(6)	655.9	24	2.483	11.084	5.55	351.163
	(7)	661.1	24	2.285	9.808	4.924	326.003

TABLEAU 4.2 Comparaison des scénarios pour les problèmes de type B avec 4 dépôts

Nbr Tâches	Numéro Scénario	Nbr Noeuds	Nbr Coupes	Temps $B(H)$(s)	Temps CycImp(s)	Temps de liftage(s)	Temps TotalCPU(s)
350	(1)	113	34	1.24	5.469	2.912	118.535
	(2)	90.3	34	1.311	5.77	2.966	106.18
	(3)	89.4	34	1.214	5.438	3.071	95.049
	(4)	135.3	34	1.125	5.175	3.055	107.409
	(5)	98.2	17	0.904	4.407	1.75	79.681
	(6)	80.6	17	1.03	4.531	1.832	82.338
	(7)	84.6	17	0.989	4.194	1.685	**69.076**
400	(1)	391	40	1.99	8.607	4.644	237.753
	(2)	385.7	40	1.716	7.98	4.351	230.379
	(3)	412.8	40	1.749	8.222	4.612	229.146
	(4)	294.1	40	1.582	7.435	4.525	235.166
	(5)	329.9	20	1.325	6.567	2.727	226.46
	(6)	374.3	20	1.403	6.431	2.69	260.82
	(7)	335.2	20	1.26	5.631	2.446	**207.291**
450	(1)	250.4	44	2.535	11.54	6.154	328.86
	(2)	192.1	44	2.24	10.224	5.97	314.534
	(3)	235.9	44	2.722	12.287	6.251	356.941
	(4)	202.7	44	2.5	11.733	6.358	325.751
	(5)	229	22	2.008	9.767	3.869	306.959
	(6)	234.6	22	2.018	9.276	3.878	304.186
	(7)	205.3	22	1.708	7.771	3.404	**235.004**
500	(1)	863	50	4.951	20.358	8.977	496.353
	(2)	816.5	50	4.024	18.154	8.646	**488.289**
	(3)	914.3	50	3.708	17.334	9.009	594.347
	(4)	636.1	50	3.802	17.599	9.356	663.665
	(5)	898.7	24	2.742	13.079	5.341	569.92
	(6)	663.6	24	2.628	11.458	5.153	505.645
	(7)	667.7	24	2.507	11.142	4.856	490.408
550	(1)	1173.9	54	5.82	26.71	12.077	1263.22
	(2)	1338.6	54	4.723	21.109	11.371	**935.408**
	(3)	1427.6	54	4.583	21.721	12.105	1067.053
	(4)	1931.7	54	4.29	19.503	11.718	1151.385
	(5)	1830.1	27	3.792	17.647	7.135	1249.793
	(6)	1798.1	27	4.108	18.278	7.383	1213.691
	(7)	1755.5	27	3.914	17.507	6.929	1224.502

350, 400, 450, 500, 550 avec 4 dépôts. Nous avons aussi testé des problèmes de type A et B de 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600 et 700 tâches avec 6 dépôts. Les résultats des tests sont présentés dans les tableaux 4.3, 4.4, 4.5 et 4.6.

Nous remarquons qu'il existe seulement 3 cas (les problèmes de type A de 100 et 450 tâches avec respectivement 4 et 6 dépôts et le problème de type B de 300 tachês avec 4 dépôts) où la méthode de « Branch-and-Cut » de CPLEX résout plus rapidement les problèmes et avec moins de noeuds dans l'arbre de branchement que nos 3 scénarios, ce qui représente moins de 7% des cas étudiés. Dans 42% des cas, le temps de résolution de la méthode de « Branch-and-Cut » de CPLEX est inférieur à celui de nos scénarios, mais pas forcément avec moins de noeuds. Par contre, dans 13% des cas présentés, son arbre de branchement contient moins de noeuds (mais dans certains problèmes le temps de résolution est supérieur à celui d'au moins un des 3 scénarios). Le scénario 7 fournit un temps de résolution inférieur à celui de la méthode « Branch-and-Cut » de CPLEX dans 44% des problèmes présentés. Dans 58% des cas, le scénario 7 explore moins de noeuds que la méthode « Branch-and-Cut » de CPLEX, et 13 cas sur 43 (ce qui représente 30% des cas) sont résolus en moins de temps et avec moins de noeuds dans l'arbre de branchement.

Pour chacun des scénarios 2 et 5, 23% des cas possèdent moins de noeuds dans leurs arbres de branchement et leurs temps de résolution sont inférieurs à ceux de la méthode « Branch-and-Cut » de CPLEX. Cette dernière est moins rapide que les scénarios 2 et 5 dans, respectivement, 33% et 28% des cas. Par contre, elle possède un arbre de branchement plus grand que les scénarios 2 et 5 dans, respectivement, 65% et 72% des cas présentés dans les tableaux 4.3, 4.4, 4.5 et 4.6.

4.7 Comparaison avec un algorithme produisant plus de coupes

Nous avons remarqué en effectuant les tests précédents que lorsque les paramètres d'ajout des coupes sont à 0, CPLEX n'ajoute pas beaucoup de coupes, et dans certains cas, aucune coupe n'est ajoutée; nous avons donc décidé d'augmenter légèrement les

TABLEAU 4.3 Comparaison des scénarios 2, 5 et 7 et du « Branch-and-Cut » de CPLEX pour des problèmes de type A avec 4 dépôts

Nbr Tâches	Numéro Scénario	Nbr Noeuds	Temps TotalCPU(s)	Diff Nbr Noeuds(%)	Diff Temps TotalCPU(%)
100	(CPLEX)	0	0.507		
	(2)	0.165	0.578		14.00
	(5)	0.161	0.572		12.82
	(7)	0	0.552	0	8.87
150	(CPLEX)	2	2.089		
	(2)	0.749	2.813	-62.55	34.65
	(5)	0.738	2.662	-63.10	27.42
	(7)	1.9	2.476	-5.00	18.52
200	(CPLEX)	8.8	6.298		
	(2)	2.121	8.298	-83.48	31.75
	(5)	2.141	7.28	-75.67	15.59
	(7)	7.7	6.61	-12.50	4.95
250	(CPLEX)	18.3	15.984		
	(2)	4.824	19.839	-73.63	24.11
	(5)	4.809	16.27	-73.63	1.78
	(7)	13.1	15.097	-28.41	**-5.54**
300	(CPLEX)	29.9	29.388		
	(2)	8.718	32.185	-21.18	9.51
	(5)	8.317	28.044	-72.18	-4.57
	(7)	31.1	25.207	4.01	**-14.22**
350	(CPLEX)	24.4	51.493		
	(2)	14.548	53.346	-40.37	3.59
	(5)	15.179	57.729	-37.79	12.11
	(7)	40.3	58.097	65.16	12.82
400	(CPLEX)	106.2	101.256		
	(2)	23.714	128.588	-77.67	26.99
	(5)	24.286	131.448	-77.13	29.81
	(7)	167.8	123.578	58.00	22.04
450	(CPLEX)	142	210.646		
	(2)	32.941	197.899	-76.80	-6.05
	(5)	36.868	180.845	-74.03	-14.14
	(7)	145.9	149.591	2.74	**-28.98**
500	(CPLEX)	788.7	399.07		
	(2)	47.41	284.176	-93.98	**-28.79**
	(5)	51.849	300.283	-93.42	-24.75
	(7)	661.1	326.003	-16.17	-18.30

TABLEAU 4.4 Comparaison des scénarios 2, 5 et 7 et du « Branch-and-Cut » de CPLEX pour des problèmes de type B avec 4 dépôts

Nbr Tâches	Numéro Scénario	Nbr Noeuds	Temps TotalCPU(s)	Diff Nbr Noeuds(%)	Diff Temps TotalCPU(%)
100	(CPLEX)	0.5	0.839		
	(2)	0.5	0.962	0.00	14.66
	(5)	0.3	0.927	-40.00	10.49
	(7)	0	0.78	-100	**-7.03**
150	(CPLEX)	7.6	2.756		
	(2)	4.2	3.788	-44.74	37.45
	(5)	5.3	3.577	-30.26	29.79
	(7)	8.7	3.313	14.47	20.21
200	(CPLEX)	27.8	8.669		
	(2)	32.6	11.243	17.27	29.69
	(5)	26.3	11.683	-5.40	34.77
	(7)	42.2	10.811	51.8	24.71
250	(CPLEX)	34.2	18.881		
	(2)	35.1	28.219	2.63	49.46
	(5)	24.3	23.722	-28.95	25.64
	(7)	32.2	21.673	-5.85	14.79
300	(CPLEX)	47.8	34.739		
	(2)	51.4	50.398	7.53	45.08
	(5)	50.5	54.567	5.65	57.08
	(7)	62.8	53.777	31.38	54.8
350	(CPLEX)	92.2	81.701		
	(2)	90.3	106.18	-2.06	29.96
	(5)	98.2	79.681	6.51	-2.47
	(7)	84.6	69.076	-8.24	**-15.45**
400	(CPLEX)	375.1	256.024		
	(2)	385.7	230.379	2.83	-10.02
	(5)	329.9	226.46	-12.05	-11.55
	(7)	335.2	207.291	-10.64	**-19.03**
450	(CPLEX)	246.2	303.776		
	(2)	192.1	314.534	-21.97	3.54
	(5)	229	306.959	-6.99	1.05
	(7)	205.3	235.004	-16.61	**-22.64**
500	(CPLEX)	596.6	536.857		
	(2)	816.5	488.289	36.86	**-9.05**
	(5)	898.7	569.92	50.64	6.16
	(7)	667.7	490.408	11.92	-8.65
550	(CPLEX)	1687.6	1113.857		
	(2)	1338.6	935.408	-20.68	**-16.02**
	(5)	1830.1	1249.793	8.44	12.20
	(7)	1755.5	1224.502	4.02	9.93

TABLEAU 4.5 Comparaison des scénarios 2, 5 et 7 et du « Branch-and-Cut » de CPLEX pour des problèmes de type A avec 6 dépôts

Nbr Tâches	Numéro Scénario	Nbr Noeuds	Temps TotalCPU(s)	Diff Nbr Noeuds(%)	Diff Temps TotalCPU(%)
100	(CPLEX)	1.7	1.662		
	(2)	1.1	1.721	-35.29	3.55
	(5)	1.5	1.768	-11.76	6.38
	(7)	0.8	1.651	-52.94	**-0.66**
150	(CPLEX)	2.3	4.213		
	(2)	2	4.935	-13.04	17.14
	(5)	2	4.949	-13.04	17.47
	(7)	2	4.437	-13.04	5.32
200	(CPLEX)	14.6	14.978		
	(2)	10.9	17.32	-25.34	15.64
	(5)	11.6	16.517	-20.55	10.28
	(7)	11.7	15.231	-19.86	1.69
250	(CPLEX)	75.2	43.262		
	(2)	69.6	48.169	-7.45	11.34
	(5)	55.5	57.722	-26.20	33.42
	(7)	55.6	53.969	-26.069	24.75
300	(CPLEX)	47.4	69.165		
	(2)	37.5	68.816	-20.89	**-0.50**
	(5)	54.4	99.046	14.77	43.20
	(7)	64.7	97.842	36.50	41.46
350	(CPLEX)	162.1	206.01		
	(2)	143.3	265.695	-11.60	28.97
	(5)	156.1	224.767	-3.70	9.10
	(7)	152.4	222.827	-5.98	8.16
400	(CPLEX)	329.4	444.045		
	(2)	408.1	480.199	23.89	8.14
	(5)	396.8	448.683	20.46	1.04
	(7)	366.3	389.468	11.20	**-12.29**
450	(CPLEX)	521.7	576.292		
	(2)	635.1	755.88	21.74	31.16
	(5)	586.4	797.201	12.40	38.33
	(7)	619.8	710.064	18.80	23.21
500	(CPLEX)	1729.3	1478.704		
	(2)	1183.8	1025.21	-31.54	-30.67
	(5)	989.9	1151.869	-42.76	-22.10
	(7)	1026	943.028	-40.67	**-36.23**
550	(CPLEX)	2867.7	2938.412		
	(2)	1888.6	2277.713	-34.14	**-22.48**
	(5)	2355.8	2844.806	-17.85	-3.19
	(7)	2241.6	3000.765	-21.83	2.12
600	(CPLEX)	4789.9	5284.238		
	(2)	6381.5	6288.877	33.23	19.01
	(5)	5295.1	4811.503	10.55	**-8.95**
	(7)	7357	5736.38	53.59	8.56
700	(CPLEX)	>61155.4	> 61663.217		
	(2)	55494	51944.523	< -16.12	< -15.76
	(5)	49157.5	49935.892	< -25.69	<-19.02

TABLEAU 4.6 Comparaison des scénarios 2, 5 et 7 et du « Branch-and-Cut » de CPLEX pour des problèmes de type B avec 6 dépôts

Nbr Tâches	Numéro Scénario	Nbr Noeuds	Temps TotalCPU(s)	Diff Nbr Noeuds(%)	Diff Temps TotalCPU(%)
100	(CPLEX)	2.3	1.843		
	(2)	2	2.677	-13.04	45.25
	(5)	2	2.697	-13.04	46.34
	(7)	4	2.161	73.91	17.25
150	(CPLEX)	7.7	5.528		
	(2)	5.9	6.667 8	-23.38	20.60
	(5)	7.4	6.864	-3.90	24.17
	(7)	5.6	5.94	-27.27	7.45
200	(CPLEX)	12	13.833		
	(2)	11.1	18.671	-7.50	34.97
	(5)	12.4	16.269	3.33	17.61
	(7)	10.8	14.956	-10.00	8.12
250	(CPLEX)	61.3	45.305		
	(2)	44.7	61.163	-27.08	34.92
	(5)	47.9	50.723	-21.86	11.96
	(7)	48.7	49.658	-20.55	9.61
300	(CPLEX)	190.9	130.457		
	(2)	169.1	105.343	-11.42	-19.25
	(5)	204.2	140.025	6.97	7.33
	(7)	202	102.125	5.81	**-21.72**
350	(CPLEX)	102.1	171.573		
	(2)	116.7	190.21	14.30	10.86
	(5)	104.7	171.93	2.55	0.21
	(7)	116.2	163.261	13.81	**-4.84**
400	(CPLEX)	986.5	640.595		
	(2)	1183.3	770.233	19.95	20.24
	(5)	719	675.661	-27.12	5.47
	(7)	827.7	618.524	-16.10	**-3.45**
450	(CPLEX)	1166.6	1081.403		
	(2)	1197.7	1036.485	2.67	**-4.15**
	(5)	1512.2	1273.825	29.62	17.79
	(7)	1530.5	1107.27	31.19	2.39
500	(CPLEX)	1922.4	2028.129		
	(2)	1845.2	1900.028	-4.02	-6.32
	(5)	1834.3	1842.659	-4.58	**-9.14**
	(7)	1810.3	2138.84	-5.83	5.46
550	(CPLEX)	3342.4	2895.346		
	(2)	3478.8	3400.365	4.08	17.44
	(5)	2599.9	3023.346	-22.21	4.42
	(7)	2455.1	2471.864	-26.55	**-14.63**
600	(CPLEX)	7888.9	7246.676		
	(2)	8260.2	7206.916	4.71	-0.55
	(5)	7222.2	6638.095	-8.45	-8.40
	(7)	4874.5	5362.033	-38.21	**-26.01**
700	(CPLEX)	56289.5	63384.179		
	(2)	35288.1	36675.309	-37.31	**-42.14**

paramètres à 1, c'est-à-dire, que nous demandons à CPLEX de chercher modérément des coupes qui peuvent être efficaces. Et nous avons comparé les résultats obtenus avec le scénario (2') suivant.

- $SeuilVio = 0.02$
- Si le nombre de coupes ajoutées est égal à $Max\{10, n/10\}$, alors cesser defaire appel à la procédure de séparation des inégalités valides.
- Si un sommet de $B(H)$ appartient à un trou impair détecté, alors ne pas le prendre comme source pour la recherche du plus court chemin.
- Les paramètres d'ajout des coupes sont fixés à -1, c'est-à-dire que seulement les coupes qui sont générées par notre méthode sont ajoutées.

Nous avons désiré comparer les résultats de CPLEX avec un scénario qui ajoute beaucoup de coupes, tout en sachant qu'il met plus de temps que les autres scénarios déjá testés ; l'analyse des résultats présentés dans les tableaux 4.7 et 4.8 nous permet de constater que :

1. le nombre de coupes ajoutées par notre scénario est toujours supérieur à celui de la méthode de « Branch-and-Cut » appliquée par CPLEX ;
2. les temps CPU totaux de résolution et les temps CPU au noeud racine de notre scénario sont toujours inférieurs à ceux enregistrés par la méthode de « Branch-and-Cut » appliquée par CPLEX ;
3. à partir des problèmes de 200 tâches, le nombre de noeuds de l'arbre de branchement obtenu en appliquant notre scénario est inférieur au nombre de noeuds de l'arbre de CPLEX ;
4. par contre pour le « GAP » enregistré au noeud racine, dans certains cas le « GAP » donné par la méthode de « Branch-and-Cut » appliquée par CPLEX est inférieur au « GAP » donné par notre scénario, et dans d'autres cas c'est l'inverse qui se produit.

Il en résulte que, bien que notre méthode ajoute plus de coupes que CPLEX, dans tous les cas son temps CPU de résolution au noeud racine et son temps CPU total de résolution sont plus petits que ceux de CPLEX. Dans la pluspart des cas, le nombre de noeuds de l'arbre de branchement est aussi plus petit que celui de CPLEX, et dans 47.5% des cas, le GAP enregistré au noeud racine est meilleur que celui de CPLEX. Ceci traduit l'efficacité des coupes générées par notre algorithme.

4.8 Comparaison du scénario (2′) avec la méthode de séparation et évaluation progressive

Notre dernière comparaison est effectuée entre le scénario (2′) (présenté dans la section précédente) et la méthode de séparation et évaluation progressive (sans coupes) appliquée par CPLEX. Les résulats montrent qu'en moyenne le nombre de noeuds de l'arbre de branchement des problèmes traités par le scénario (2′) est inférieur à celui de la méthode de séparation et évaluation progressive, nous avons une diminution de 7.93 % pour les instances avec 4 dépôts, et une diminution de 12.33 % pour les instances avec 6 dépôts, par rapport au nombre de noeuds enregistré par CPLEX. Le temps CPU total de résolution du scénario (2′) est, en moyenne, supérieur à celui de la méthode de séparation et évaluation progressive de 18.06% pour les instances avec 4 dépôts et de seulement 10.33% pour les instances avec 6 dépôts. Nous remarquons aussi que pour les instances de moins de 350 tâches, la moyenne des différences entre les temps CPU totaux de résolution est de moins de 8 secondes. Mais ce qui est remarquable, c'est que pour les problèmes ayant entre 450 et 550 tâches, sauf pour les instances de 450 de type A avec 4 et 6 dépôts et 550 de type B avec 6 dépôts, le temps total CPU de résolution enregistré par le scénario (2′) est meilleur que celui de la méthode de séparation et évaluation progressive. Il est important de noter que le scénario (2′) n'est pas le meilleur de nos scénarios testés précédemment.

TABLEAU 4.7 Comparaison entre le scénario (2′) et CPLEX pour des problèmes avec 4 dépôts

Nbr Tâches	Numéro Scénario	NbrCoupes Racine	« GAP » Racine	Temps Racine	Nbr Noeuds	Nbr Coupes	Temps Total CPU(s)
100A	(2′)	3.4	0.000636	**0.447**	0	**3.4**	**0.442**
	(CPLEX)	1	0.000636	0.602	0	1	0.602
100B	(2′)	4	**0.002597**	**0.541**	**0.2**	**4**	**0.543**
	(CPLEX)	0.8	0.003732	0.601	0.6	0.8	0.614
150A	(2′)	6.9	**0.003303**	**2.195**	2.7	**6.9**	**2.338**
	(CPLEX)	2.2	0.00571	2.735	**1.4**	2.2	2.807
150B	(2′)	11.2	0.252571	**2.531**	**4.1**	**11.2**	**2.951**
	(CPLEX)	5.7	**0.007212**	4.243	8.2	6	4.73
200A	(2′)	14	0.006623	**6.298**	**7.4**	**14**	**7.083**
	(CPLEX)	4.8	**0.005847**	9.37	9.9	4.8	10.607
200B	(2′)	18	0.016366	**8.083**	**32.6**	**18**	**11.2**
	(CPLEX)	7.4	**0.015872**	16.19	38.7	7.8	20.148
250A	(2′)	21.1	0.010131	**15.051**	20.4	**21.1**	**17.419**
	(CPLEX)	5	**0.006753**	22.259	**12.9**	5	24.221
250B	(2′)	24	0.017916	**18.468**	**35.1**	**24**	**24.238**
	(CPLEX)	10.6	**0.010999**	30.997	42.9	10.7	38.096
300A	(2′)	22	0.008827	**22.377**	**38.2**	**22**	**28.697**
	(CPLEX)	8.9	**0.004957**	37.522	40.2	9.3	45.863
300B	(2′)	30	0.049103	**37.982**	**51.4**	**30**	**50.825**
	(CPLEX)	5.7	**0.026348**	55.598	89.4	6.3	73.858
350A	(2′)	30.4	**0.007169**	**44.263**	**37.8**	**30.6**	**55.876**
	(CPLEX)	5.1	0.014904	76.217	44	6.6	87.356
350B	(2′)	34	0.050072	**71.139**	**90.3**	**34**	**102.208**
	(CPLEX)	7.9	**0.017233**	99.536	171.4	8.8	139.295
400A	(2′)	39.9	**0.023693**	**87.007**	**101.3**	**40**	**130.517**
	(CPLEX)	6.8	0.1239	140.525	197.7	8.9	200.978
400B	(2′)	40	0.147829	**144.397**	**386.1**	**40**	**230.716**
	(CPLEX)	8.1	**0.137332**	189.235	418.1	9	292.22
450A	(2′)	39.6	0.029885	**139.446**	**139.3**	**39.6**	**200.118**
	(CPLEX)	6.3	**0.013163**	170.998	325.3	7.2	255.071
450B	(2′)	44	0.703807	**181.819**	**194.4**	**44**	**315.394**
	(CPLEX)	10.9	**0.42222**	280.797	561.8	11.2	544.957
500A	(2′)	49.6	0.109904	**175.726**	**410.7**	**49.6**	**295.862**
	(CPLEX)	6.6	**0.108862**	293.116	1069.8	7.4	630.296
500B	(2′)	48.9	**0.026017**	**239.427**	**818**	**50**	**527.03**
	(CPLEX)	7.8	0.151627	392.474	1130.5	8.8	945.586
550A	(2′)	54	0.332495	**282.502**	**266.9**	**54**	**417.871**
	(CPLEX)	8.3	**0.254144**	421.67	708.7	9.4	727.597
550B	(2′)	54	**0.073514**	**402.767**	**1338.6**	**54**	**973.302**
	(CPLEX)	8.6	0.080886	576.918	3501.7	9.8	1675.908

TABLEAU 4.8 Comparaison entre le scénario (2′) et CPLEX pour des problèmes avec 6 dépôts

Nbr Tâches	Numéro Scenario	NbrCoupes Racine	GAP Racine	Temps Racine	Nbr Noeuds	Nbr Coupes	Temps TotalCPU(s)
100A	(2′)	6	**0.004828**	**1.207**	**0.9**	**6**	**1.232**
	(CPLEX)	1	0.010237	1.548	1.5	1	1.569
100B	(2′)	7.5	**0.009637**	**1.565**	2.7	**7.5**	**1.712**
	(CPLEX)	2.5	0.013766	2.004	**1.4**	2.5	2.036
150A	(2′)	8.8	**0.003375**	**4.075**	2	**8.8**	**4.319**
	(CPLEX)	5	0.006705	6.493	**1.5**	5	6.676
150B	(2′)	9.8	0.013513	**5.821**	5.9	**9.8**	**6.26**
	(CPLEX)	4.7	**0.006253**	9.121	**4.2**	4.7	9.628
200A	(2′)	18.9	**0.014447**	**14.576**	**10.9**	**18.9**	**16.99**
	(CPLEX)	9.7	0.02354	26.607	25	9.9	32.147
200B	(2′)	18	0.01932	**14.837**	**10.2**	**18**	**17.571**
	(CPLEX)	5.9	**0.009772**	23.481	20.3	6.1	26.83
250A	(2′)	24	**0.01121**	**32.016**	**69.6**	**24**	**44.684**
	(CPLEX)	8.6	0.018114	55.703	128.1	9.5	76.351
250B	(2′)	24	**0.030353**	**40.283**	**44.7**	**24**	**54.938**
	(CPLEX)	8.2	0.185423	67.672	71.8	9	86.343
300A	(2′)	30	**0.006051**	**52.964**	**37.5**	**30**	**66.029**
	(CPLEX)	7.8	0.017773	102.652	72.3	8	127.527
300B	(2′)	26.6	**0.034197**	**67.498**	**169**	**26.6**	**117.221**
	(CPLEX)	9.7	0.043203	122.002	331.7	10.1	191.051
350A	(2′)	34	0.718799	**154.633**	**144.1**	**34**	**253.428**
	(CPLEX)	8.3	**0.422696**	286.068	307.3	10.2	427.213
350B	(2′)	34	**0.036259**	**127.024**	**116.7**	**34**	**191.151**
	(CPLEX)	9.1	0.178866	204.658	163.7	10.4	282.329
400A	(2′)	40	0.697029	298.864	**408.1**	**40**	**502.43**
	(CPLEX)	7	**0.032169**	**295.441**	716.8	8.3	560.195
400B	(2′)	40	0.550774	**322.252**	**1183.3**	**40**	**836.476**
	(CPLEX)	10.2	**0.313888**	428.891	2105.8	11.6	1217.129
450A	(2′)	33.7	0.361191	**386.316**	**635.1**	**35.2**	**762.264**
	(CPLEX)	6.6	**0.028294**	428.246	1173.9	7.2	1013.727
450B	(2′)	44	**10.071091**	**397.16**	**1197.7**	**44**	**1015.649**
	(CPLEX)	6.7	10.63364	549.534	2732.9	6.8	2093.584
500A	(2′)	47.4	**0.019144**	**445.242**	**1183.8**	**50**	**1022.303**
	(CPLEX)	8.6	0.130757	660.837	2176.8	9.3	1981.51
500B	(2′)	50	1.13463	**714.702**	**1845**	**50**	**1974.522**
	(CPLEX)	8.5	**0.281287**	777.844	2694.9	9	2308.57
550A	(2′)	54	**0.307153**	**844.798**	**1888.7**	**54**	**2408.574**
	(CPLEX)	6.7	1.107968	1286.66	9301.4	7.1	7729.052
550B	(2′)	54	**1.056797**	**971.68**	**3478**	**54**	**3286.411**
	(CPLEX)	8	1.284372	1431.34	10342.5	8.8	7443.932

TABLEAU 4.9 Comparaison entre le scénario (2′) et un algorithme standard de séparation et évaluation progressive pour des problèmes avec 4 dépôts

Nbr Tâches	Numéro Scenario	Nbr Noeuds	Temps Total CPU(s)	Diff Nbr Noeuds (%)	Diff Temps Total CPU (%)
100A	(2′)	0	0.442	-100	5.24
	(SEP)	0.1	0.42		
100B	(2′)	0.2	0.543	-60	3.23
	(SEP)	0.5	0.526		
150A	(2′)	2.7	2.338	17.39	26.65
	(SEP)	2.3	1.846		
150B	(2′)	4.1	2.951	-29.31	22.19
	(SEP)	5.8	2.415		
200A	(2′)	7.4	7.083	-18.68	28.27
	(SEP)	9.1	5.522		
200B	(2′)	32.6	11.2	10.51	15.45
	(SEP)	29.5	9.701		
250A	(2′)	20.4	17.419	11.48	5.11
	(SEP)	18.3	16.572		
250B	(2′)	35.1	24.238	3.54	26.5
	(SEP)	33.9	19.16		
300A	(2′)	38.2	28.697	34.04	24.76
	(SEP)	28.5	23.002		
300B	(2′)	51.4	50.825	17.89	64.55
	(SEP)	43.6	30.888		
350A	(2′)	37.8	55.876	29.9	28.59
	(SEP)	29.1	43.454		
350B	(2′)	90.3	102.208	-17.38	57.31
	(SEP)	109.3	64.973		
400A	(2′)	101.3	130.517	-11.22	10.05
	(SEP)	114.1	118.601		
400B	(2′)	386.1	230.716	23.32	29.66
	(SEP)	313.1	177.934		
450A	(2′)	139.3	200.118	-25.98	43.59
	(SEP)	188.2	139.367		
450B	(2′)	194.4	315.394	-34.08	-3.5
	(SEP)	294.9	326.834		
500A	(2′)	410.7	295.862	-46.32	-17.35
	(SEP)	765.1	357.95		
500B	(2′)	818	527.03	45.4	-1.58
	(SEP)	562.6	535.496		
550A	(2′)	266.9	417.871	-30.53	-2.51
	(SEP)	384.2	428.645		
550B	(2′)	1338.6	973.302	-15.85	-4.92
	(SEP)	1590.8	1023.674		

TABLEAU 4.10 Comparaison entre le scénario (2′) et un algorithme standard de séparation et évaluation progressive pour des problèmes avec 6 dépôts

Nbr Tâches	Numéro Scenario	Nbr Noeuds	Temps Total CPU(s)	Diff Nbr Noeuds (%)	Diff Temps Total CPU (%)
100A	(2′)	0.9	1.232	-55	10.89
	(SEP)	2	1.111		
100B	(2′)	2.7	1.712	28.57	23.64
	(SEP)	2.1	1.383		
150A	(2′)	2	4.319	-28.57	6.56
	(SEP)	2.8	4.053		
150B	(2′)	5.9	6.26	7.27	29.29
	(SEP)	5.5	4.842		
200A	(2′)	10.9	16.99	-46.04	13.29
	(SEP)	20.2	14.997		
200B	(2′)	10.2	17.571	-29.66	22.51
	(SEP)	14.5	14.342		
250A	(2′)	69.6	44.684	-2.52	7.51
	(SEP)	71.4	41.562		
250B	(2′)	44.7	54.938	-33.18	19.41
	(SEP)	66.9	46.008		
300A	(2′)	37.5	66.029	-18.3	-6.2
	(SEP)	45.9	70.393		
300B	(2′)	169	117.221	-11.19	2.05
	(SEP)	190.3	114.861		
350A	(2′)	144.1	253.428	-3.35	20.21
	(SEP)	149.1	210.826		
350B	(2′)	116.7	191.151	33.68	23.22
	(SEP)	87.3	155.128		
400A	(2′)	408.1	502.43	9.47	24.13
	(SEP)	372.8	404.757		
400B	(2′)	1183.3	836.476	24.06	28.27
	(SEP)	953.8	652.145		
450A	(2′)	635.1	762.264	4.3	23.32
	(SEP)	608.9	618.14		
450B	(2′)	1197.7	1015.649	-36.23	-16.63
	(SEP)	1878.1	1218.298		
500A	(2′)	1183.8	1022.303	-17.6	-14.9
	(SEP)	1436.6	1201.238		
500B	(2′)	1845	1974.522	-27.08	-8.17
	(SEP)	2530.2	2150.235		
550A	(2′)	1888.7	2408.574	-35.18	-7.64
	(SEP)	2913.8	2607.713		
550B	(2′)	3478	3286.411	-10.06	5.87
	(SEP)	3866.9	3104.186		

CONCLUSION

Bien que nous soyons incapables de conclure que l'un des scénarios est meilleur ou pire que le « Branch-and-Cut » de CPLEX, nous pouvons dire que notre algorithme est prometteur. Nous avons tenté d'introduire une méthode pour fixer les variables inspirée de celle de Hadjar, Marcotte et Soumis (2006), mais malheureusement CPLEX ne permet pas la suppression des variables dans un arbre de branchement, ou plus précisement il ne sait pas comment continuer la résolution du problème après la suppression des variables. Il est sans doute possible de le faire mais en gérant le branchement tout au long de la résolution du problème ; ceci prendra-t-il plus de temps ou moins ? Seule l'expérience peut nous donner une réponse. Il est aussi possible de résoudre la relaxation, de supprimer certaines variables, en appliquant une méthode pour fixer les variables, et ensuite de résoudre le problème avec notre algorithme, mais nous avons besoin d'une solution heuristique entière initiale. Nous avons l'intention d'implémenter et de tester cette méthode.

APPENDICE A

LES FONCTIONS ET PARAMÈTRES POUR L'AJOUT DES COUPES

CPLEX permet l'ajout des coupes globales (des coupes valables pour l'ensemble des noeuds de l'arbre de branchement) à l'aide de la fonction « CPXcutcallbackadd ». Il est aussi possible d'ajouter des coupes locales, c'est-à-dire des coupes valables pour un noeud donné de l'arbre de branchement; s'il existe des coupes qui peuvent être ajoutées, nous utilisons la fonction « CPXcutcallbackaddlocal », qui ajoute les coupes au programme linéaire du sous-problème associé au noeud courant. Les coupes ajoutées sont valables pour le sous-arbre engendré par le noeud courant.

Nous devons écrire une procédure, nommée par exemple cutcallback ; dans cette dernière, nous déterminons nos coupes ; ensuite nous faisons appel à la fonction « CPXcutcallbackaddlocal » (ou à « CPXcutcallbackadd », si nous voulons ajouter des coupes globales), qui effectuera l'ajout des coupes au noeud courant.

Pour que ILOG CPLEX appelle « cutcallback » à chaque noeud de l'arbre de branchement, nous avons besoin de faire appel à la fonction « CPXsetcallbackfunc », avant l'appel de CPXmipopt. Mais avant tout, nous devons fixer certains paramètres et cela en utilisant la fonction « CPXsetintparam ».

- int CPXPUBLIC CPXsetintparam(CPXENVptr env, int whichparam, int newvalue)

  env : pointeur à l'environnement CPLEX, qui est retourné par CPXopenC-

PLEX

Afin d'assurer que les variables du modèle originel seront exprimées linéairement en fonction des variables du modèle résolu, nous fixons « whichparam » à « CPX_PARAM_PRELINEAR » et « newvalue »à 0.

Si nous désirons que l'ajout des coupes se fasse sur le problème originel, nous devons faire un autre appel de la fonction, en fixant « whichparam » à « CPX_PARAM_MIPCBREDLP » et « newvalue » à « CPX_OFF ».

- int CPXPUBLIC CPXsetcutcallbackfunc(CPXENVptr env, int (CPXPUBLIC *cutcallback) (CALLBACK_CUT_ARGS), void *cbhandle)

  env : pointeur à l'environnement CPLEX, qui est retourné par CPXopenCPLEX

  cutcallback : un pointeur à la procédure dans laquelle nous avons défini nos coupes et la manière de les ajouter.

  cbhandle : un pointeur à la structure qui contient les données nécessaires pour l'ajout des coupes. Ce pointeur passera par « cutcallback ».

  La fonction « CPXsetcutcallbackfunc » permet l'appel de la fonction « cutcallback » au niveau de chaque noeud de l'arbre de branchement, que son programme linéaire possède une solution optimale non-entière avec une valeur optimale de l'objectif.

- int CPXPUBLIC cutcallback(CPXENVptr env, void *cbdata, int wherefrom, void *cbhandle, int *useraction_p)

  env : pointeur à l'environnement CPLEX, qui est retourné par CPXopenCPLEX

  cbdata : un pointeur à passer de la procédure d'optimisation à cutcallback et identifiant le problème à optimiser.

  wherefrom : une valeur entière indiquant à quel instant cette fonction doit être appelée dans la procédure d'optimisation. Sa valeur est « CPX_CALLBACK_MIP_CUT ».

cbhandle : un pointeur pour utiliser les informations propres au problème à étudier.

useraction_p : un pointeur entier indiquant l'action que ILOG CPLEX doit prendre pour l'exécution de cutcallback.

- int CPXPUBLIC CPXcutcallbackaddlocal(CPXENVptr env, void *cbdata, int wherefrom, int nzcnt, double rhs, int sense, const int *cutind, const double *cutval)

  env : pointeur à l'environnement CPLEX, qui est retourné par CPXopenCPLEX

  cbdata : l'indicateur à passer à cutcallback

  wherefrom : une valeur entière, indiquant où cutcallback est appelée. Ce paramètre peut prendre la valeur du « wherefrom » passé à cutcallback.

  nzcnt : une valeur entière indiquant le nombre de coefficients dans la coupe.

  rhs : une valeur indiquant le second membre de la coupe.

  sense : une variable indiquant le sens de la coupe.

  cutind : un vecteur contenant les indices des coefficients de la coupe.

  cutval : un vecteur contenant les valeurs des coefficients de la coupe.

BIBLIOGRAPHIE

Aardal K., Hoesel S.V. (1996), *Polyhedral Techniques in Combinatorial Optimization I : Theory*, Staistica Neerlandica, 50, pp. 3-26

Aardal K., Hoesel S.V. (1999), *Polyhedral Techniques in Combinatorial Optimization II : Computations and application*, Staistica Neerlandica, **53** pp. 129-178

Balas E. (1965), *An additive algorithm for solving linear programs with zero-one variables*, The Journal of the Operations Research Society of America, pp. 517-546

Bertossi A.A., Carraresi P., Gallo G. (1987), *On some matching problems arising in vehicle scheduling models*, Networks **17** pp. 271-281.

Bouzgarrou M.E. (1998), *Parallélisation de la méthode du Branch-and-Cut pour résoudre le problème du voyageur de commerce*, Thèse de doctorat de l'Institut National Polytechnique de Grenoble.

Carpaneto G., Dell'Amico M., Fischetti M., Toth P. (1989),*A branch-and-bound algorithm for the multiple depot vehicle scheduling problem*, Networks, vol. 19 pp. 531-548.

Chvátal V. (1983), *Linear Programming*, W.H. Freeman and Company.

Cook W.J., Cunningham W.H., Pulleyblank W.R., Schrijver A. (1998), *Combinatorial Optimization*, J.Wiley and Sons.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2002), *Introduction à l'algorithmique*, Dunod, Paris.

Desaulniers G., Hickman M. (2007), *Public Transit*, In : G.Laporte and C.Barnhart (eds), *Tansportation*, Handbooks in Operations Research and Management Science, Elsevier Science, Amsterdam.

Dijkstra E.W. (1959), *A note on two problems in connexion with graphs*, Numerische Mathematik, 1 pp. 269-271.

Fischetti M., Dell'Amico M., Toth P. (1993), *Heuristic algorithms for the multiple depot vehicle scheduling problem*, Management Science **39** pp. 115-125.

Fischetti M., Lodi A., Martello S., Toth P. (2001), *A polyhedral approach to simplified crew scheduling and vehicle scheduling problems*, Management Science **47** 833-850.

Forbes M.A., Holt J.N., Watts A.M. (1994), *An exact algorithm for Multiple Depot Bus Scheduling*, European Journal of Operational Research **72** pp. 115-124.

Gondran M., Minoux M. (1985), *Graphes et Algorithmes*, Editions Eyrolles.

Grötschel M., Jünger M., Reinelt G. (1984), *A cutting plane algorithm for the linear ordering problem*, Operations Research, 32, pp. 1195-1220.

Grötschel M., Lovász L., Schrijver A. (1988), *Geometric Algorithms and Combinatorial Optimization*, Springer Verlag, Berlin-Heidelberg.

Hadjar A., Marcotte O., Soumis F. (2006), *A Branch-and-Cut Algorithm for the Multiple Depot Vehicle Scheduling Problem*, Operations Reserch, vol. 54, pp. 130-149.

Hartman J.C., Ralphs T.K. (2001), *Capacitated Node Routing Problems* (Preliminary Progress Report).

Land A.H., Doig A.G. (1960), *An automatic method for solving discrete programming problems*, Econometrica, **28**, pp. 97-520.

Little J.D.C., Murty G., Sweeney D.W., Karel C. (1963), *An algorithm for the traveling salesman problem*, Operations Research, Vol. 11, No. 6, pp. 972-989.

Löbel A. (1997), *Experiments with a Dantzig-Wolfe decomposition for multiple-depot vehicle scheduling problems*, Perprint SC 97-16, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

Mitchell J.E. (1999), *A Branch-and-Cut Algorithm for Combinatorial Optimization Problems*, http ://www.math.rpi.edu/~mitchj.

Nemhauser G.L., Sigismondi G. (1992), *A Strong Cutting Plane Branch-and-Bound Algorithm for Node Packing*, Operations Research Vol. 43, No. 5, pp. 443-457.

Nemhauser G.L., Wolsey L.A. (1988), *Integer and Combinatorial Optimization*, J.Wiley and Sons.

Padberg M.W., Rinaldi G. (1987), *Optimization of a 532 city symmetric traveling salesman problem by Branch-and-Cut*, Operations Research Letters, pp. 1-7.

Padberg M.W., Rinaldi G. (1991), *A Branch-and-Cut algorithm for the resolution of large-scale symmetric traveling salesman problems*, SIAM Review, vol. 33, pp. 60-100.

Pepin A.-S., Desaulniers G., Hertz A., Huisman D. (2006), *Comparison of Heuristic Approaches for the Multiple Depot Vehicle Scheduling Problem*, GERAD-HÉC Montréal G-2006-65.

Ribeiro C.C, Soumis F. (1994), *A column generation approach to the multiple depot vehicle scheduling problem*, Operations Research Vol. 42, No. 1, pp. 41-52.