Parallélisation de la méthode du "Branch and Cut" pour résoudre le problème du voyageur de commerce

Mohamed Ekbal Bouzgarrou

THÈSE

présentée par

Mohamed Ekbal BOUZGARROU

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 30 Mars 1992)

Formation Doctorale : **Recherche Opérationnelle**

Discipline : **Informatique**

Parallélisation de la méthode du "Branch and Cut" pour résoudre le problème du voyageur de commerce

---

Date de soutenance : 14 décembre 1998

Composition du jury :

Rapporteurs :	Afonso	FERREIRA
	Catherine	ROUCAIROL
Examinateurs :	Gerd	FINKE
	Denis	NADDEF (directeur de thèse)
	Denis	TRYSTRAM (président du jury)

Thèse préparée au sein du
Laboratoire de Modélisation et Calcul
(Institut d'Informatique et de Mathématiques Appliquées de Grenoble)

À mes parents
Mohamed et Saïda

Remerciements

Je tiens d'abord à remercier les membres du jury.

Madame Catherine Roucairol, Professeur de l'université de Versailles Saint Quentin en Yvelines d'avoir accepté de rapporter mon travail. Ces nombreuses remarques constructives m'ont permis de valoriser la qualité de ce document.

Monsieur Afonso Ferreira, Chargé de recherche au CNRS d'avoir accepté de rapporter ma thèse malgré son emploi du temps chargé. Ses remarques ont été d'une grande importance pour améliorer la clareté de ce manuscript.

Monsieur Gerd Finke, Professeur de l'Université Joseph Fourrier de Grenoble d'avoir accepté d'examiner mon travail.

Monsieur Denis Trystram, Professeur de l'Institut National Polytechnique de Grenoble d'avoir accepté d'examiner mon travail, et de m'avoir accueilli au sein de l'équipe Algorithmique parallèle du LMC.

Monsieur Denis Naddef, Professeur de l'Institut National Polytechnique de Grenoble d'avoir accepté de diriger mes recherches. Ses nombreux conseils, sa rigueur ainsi que sa disponibilité ont été de précieux atouts tout au long de ce travail.

Un grand merci à Olivier Briant et Phillipe Augerat qui se sont livrés à la chasse aux fautes dans ce document. Leurs conseils et leur disponibilité méritent d'être particulièrement salués.

Je voudrais aussi remercier mes collègues de bureau, Martha, François, Christophe qui a changé de compagnie et le nouveau arrivant Renaud pour l'ambiance détendue et studieuse qui y règne.

Merci également à Alfredo, Alexandre, Andrea, Benhur, Brigitte, François O., Gerson, Greg, Gustavo, Hélène, Jacques B., Jacques C., Jean Guillaume, Jean Louis, Jean Marc, Joëlle, Khadija, Marcello, Mathias, Nicolas, Phillipe W., Roberta, Thierry, Yves.

Merci aussi à mes amis Basma, Lamia, Imed, Pascal, Rym, Sami, Takoua et Wassel pour leurs encouragements et leurs aides.

Finalement, je tiens à remercier mes parents Mohamed et Saïda, ma sœur Chiraz et mon frère Ghanem pour leurs encouragements, ainsi que la famille Khalifa pour son aide et son hospitalité.

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Introduction

La résolution jusqu'à l'optimalité de problèmes d'optimisation combinatoire $\mathcal{NP}$-difficiles nécessite une mise en œuvre de méthodes de plus en plus complexes qui consomment de plus en plus de puissance de calcul. Le calcul parallèle offre la perspective de pouvoir réduire le temps de résolution de ces problèmes et permet d'espérer aborder des instances de plus grande taille.

L'objectif de notre travail est de paralléliser un algorithme de "Branch and Cut" pour pouvoir résoudre jusqu'à l'optimalité des instances du problème du voyageur de commerce symétrique.

Le problème du voyageur de commerce est parmi les problèmes les plus difficiles de la recherche opérationnelle et de l'optimisation combinatoire. Il consiste à trouver un parcours de coût minimum passsant exactement une fois par tous les sommets d'un graphe. Ce problème a été largement étudié dans plusieurs branches de la mathématique, de l'informatiques et de la recherche opérationnelle. Il possède plusieurs applications dans les domaines industriels, informatique et de distribution: problème de tournée de véhicule, problème de construction des circuits imprimés, parcours d'un robot dans un magasin, ordonnancement de tâches ayant des temps d'exécution dépendant de leurs précédences...

Quand la théorie de la complexité a été développée, le problème du voyageur de commerce fut l'un des premiers à être classé $\mathcal{NP}$-difficile par Karp en 1972. Depuis, plusieurs heuristiques et techniques algorithmiques ont été mises en place pour trouver des solutions à ce problème : la méthode du "Branch and Bound", la méthode de la relaxation Lagrangienne, les méthodes d'améliorations du type Lin Kernigham, les métas heuristiques tels que le recuit simulé ou les algorithmes génétiques, et des algorithmes qui se basent sur l'étude polyédrale du problème tels que la méthode des coupes polyédrale.

Pendant les dix dernières années plusieurs études ont été menées pour paralléliser ces méthodes et notamment la méthode du "Branch and Bound". Ces études ont permis de montrer l'apport possible du parallélisme. Cependant les tailles des

instances résolues restent en deçà des espérances.

L'émergence de la méthode du "Branch and Cut" a permis de franchir un seuil dans la taille des instances résolues du problème du voyageur de commerce. Cette méthode d'énumération implicite conjugue l'effort de plusieurs autres méthodes, essentiellement la méthode des coupes polyédrale, la méthode du "Branch and Bound" et la méthode de génération de colonnes. Elle a permis de résoudre de grandes instances du problème du voyageur de commerce, mais elle s'est avérée grande consommatrice de puissance de calcul. La parallélisation de cette méthode permettra de réduire son coût prohibitif en temps de calcul tout en abordant des instances de plus en plus difficiles du problème du voyageur de commerce.

L'objectif de notre travail est d'implémenter un algorithme parallèle, efficace et portable de "Branch and Cut" pour résoudre des instances difficiles du problème du voyageur de commerce. Les choix pris lors de la parallélisation doivent prendre en compte le caractère fortement irrégulier de l'algorithme du "Branch and Cut" ainsi que ses spécificités par rapport à d'autres méthodes classiques d'énumération implicite, notamment l'algorithme du "Branch and Bound".

Structure du document

Dans le premier chapitre, nous introduirons les méthodes de base qui composent l'algorithme du "Branch and Cut". Nous commencerons par un bref rappel sur la programmation linéaire et l'algorithme du simplexe. Puis nous décrirons la méthode du "Branch and Bound". Nous détaillerons ensuite la méthode des plans de coupes et les concepts de base de la théorie polyédrale. Nous présenterons ensuite un algorithme simplifié de "Branch and Cut". Nous terminerons ce chapitre par la méthode de génération de colonnes.

Le chapitre 2 sera consacré à une approche polyédrale de l'étude du problème du voyageur de commerce. Nous commencerons par une modélisation du problème sous la forme d'un programme linéaire en nombres entiers. Nous définirons ensuite le polytope du problème du voyageur de commerce et nous donnerons ses principales propriétés. Nous décrirons enfin quelques facettes connues du polytope du problème du voyageur de commerce.

Dans le troisième chapitre, nous détaillerons toutes les étapes de notre algorithme du "Branch and Cut" pour résoudre le problème du voyageur de commerce. Tout d'abord, nous présenterons un organigramme géneral de l'algorithme. Ensuite nous décrirons les phases d'initialisation, et de résolution du programe linéaire. Puis nous présenterons la phase de génération de coupes polyédrales pro-

venant d'heuristiques de séparations et de "pool". Nous décrirons ensuite l'étape de "pricing" et de génération de coupes, et nous montrerons les possibilités de fixation de variables pendant cette étape. Nous terminerons enfin par une présentation des stratégies de branchement et de parcours de l'arbre du "Branch and Cut".

Le chapitre 4 introduit le lecteur aux concepts de base du parallélisme. Nous présenterons ensuite dans le chapitre 5 un aperçu non exhaustif des études de parallélisation de l'algorithme du "Branch and Bound" dédié à la résolution de problème d'optimisation combinatoire. Nous rapporterons quelques exemples d'expérimentations et de plateformes de programmation de "Branch and Bound" parallèle.

Nous présenterons dans le chapitre 6, notre méthodologie de parallélisation de l'algorithme du "Branch and Cut". Nous commencerons par étudier les différents niveaux de parallélisme possibles. Nous donnerons ensuite la stratégie de contrôle de la recherche arborescente que nous avons adopté. Nous montrerons enfin les mécanismes de contrôle de communication et d'équilibrage de charge et justifierons nos choix.

Dans le dernier chapitre, nous présenterons les résultats de nos expérimentations. Nous commencerons par justifier l'irrégularité du "Branch and Cut" en montrant son comportement séquentiel. Nous présenterons ensuite les résultats que nous avons obtenus avec notre algorithme parallèle. Nous expliquerons ces résultats en insistant sur les spécificités de notre implémentation.

Nous terminerons ce document par les perspectives et les directions de nos travaux futures.

Chapitre 1

Introduction à la méthode du "Branch and Cut"

Dans ce chapitre, nous décrirons les méthodes de base qui composent l'algorithme du "Branch and Cut". Nous commencerons par un bref rappel sur la programmation linéaire et l'algorithme du simplexe. Puis nous décrirons la méthode du "Branch and Bound". Nous détaillerons ensuite la méthode des plans de coupes et les concepts de base de la théorie polyédrale. Nous présenterons alors, un algorithme simplifié de "Branch and Cut". Et nous terminerons ce chapitre par la méthode de génération de colonnes.

1.1 Programmation linéaire

Etant donnés un ensemble $E = \{e_1, \ldots, e_n\}$ et une fonction $c : E \to I\!R$, qui associe à chaque $e_i \in E$ un coût c_i. Soit $\mathcal{S} = \{S_i \subset E, i = 1 \ldots N\}$. Dans la plupart des problèmes d'optimisations combinatoires, on cherche:

$$S_i^* \text{ tel que } c(S_i^*) = \min_{1 \leq i \leq N} c(S_i) = \min_{1 \leq i \leq N} \sum_{e_j \in S_i} c_j$$

On associe à tout $S \in \mathcal{S}$ un *vecteur d'incidence* $x^S \in \{0,1\}^{|E|}$ défini par

$$x^S = \begin{cases} x_i^S = 1 & \text{si } e_i \in S \\ x_i^S = 0 & \text{si } e_i \notin S \end{cases}$$

Le problème d'optimisation est équivalent à:

$$min\{c^t x^S : S \in \mathcal{S}\} = min\{c^t x : x \in conv\{x^S : S \in \mathcal{S}\}\}$$

où $conv\{x^S : S \in \mathcal{S}\}$ est l'enveloppe convexe des vecteurs d'incidences des solutions du problème. D'après le résultat classique de Farkas, Weyl et Minkowski [Sch86], l'enveloppe convexe des vecteurs d'incidences peut être décrite par un système fini d'inéquations linéaires, i.e.

$$conv\{x^S : S \in \mathcal{S}\} = \{x : Ax \geq b\}$$

Ainsi étant donnés une matrice $A \in I\!R^{m \times n}$, un vecteur $b \in I\!R^m$ et un vecteur coût $c \in I\!R^n$, le problème revient à trouver un vecteur $x^* \in I\!R^n$ tel que:

$$c^t x^* = min\{c^t x : Ax \geq b, x \in I\!R^n\}$$

La fonction $c^t x : I\!R^n \rightarrow I\!R$ est appelée *fonction économique* (ou objectif), les inégalités dans le système $Ax \geq b$ sont appelées contraintes, x^* est appelé solution optimale du programme linéaire (P). Cette solution n'est pas nécessairement unique.

On peut résoudre efficacement ce type de problème par la méthode du simplexe (voir [Chv83]) ou par la méthode des points intérieurs (voir [LMS94]). Rappelons ici le principe de la méthode (primale et duale) du simplexe dont on utilisera plusieurs notions durant cette thèse.

Pour chaque programme linéaire (P) nommé programme linéaire primal, on associe un programme linéaire dual (D), dans lequel on doit trouver un vecteur $y^* \in I\!R^m$, tel que:

$$y^{*^t} b = max\{y^* b : A^t y = c, y \geq 0\}$$

Notons que le dual de (D) est (P). La relation entre ces deux programmes est donnée par le théorème suivant:

Théorème 1 *Etant donnés deux programmes linéaires duaux (P) et (D).*

- *Si (P) et (D) ont des solutions réalisables alors ils ont des solutions optimales, et les valeurs des solutions optimales sont égales.*

- *Si (P) n'admet pas de solution réalisable alors soit (D) n'admet pas de solution réalisable, soit (D) est non borné.*

- *Si (P) est non borné alors (D) n'admet pas de solution réalisable.*

Tout programme linéaire $min\{c^t x : Ax \geq b, x \geq 0\}$ peut être transformé pour ne comporter que des équations, en ajoutant une variable s_i appelée variable d'écart pour chaque contrainte, le programme linéaire s'écrit alors comme suit :

$$min\{c^t x : \begin{pmatrix} A & -I \end{pmatrix} \begin{pmatrix} x \\ s \end{pmatrix} = b, \begin{pmatrix} x \\ s \end{pmatrix} \in I\!R_+^{n+m}\}$$

où I est la matrice identité $m \times m$, et s est le vecteur des variables d'écarts. Notons que les deux programmes linéaires sont équivalents. On considérera dorénavant que les programmes linéaires sont écrits sous cette forme. La matrice A définissant les contraintes du programme linéaire est désormais de dimension $m \times (m+n)$, la dimension du vecteur coût c est $m+n$ tel que les m derniers éléments de c sont nuls.

Une base B de la matrice $A \in I\!R^{m \times (m+n)}$ de plein rang, est une sous-matrice de A inversible de taille $m \times m$. Les colonnes (variables) de A qui forment la matrice B sont dites *de base*, les autres N sont dites *hors base*. La colonne d de A telle que $B^{-1}d$ est le r^{ieme} vecteur unitaire, est dite de base à la r^{ieme} ligne. Pour un vecteur $x \in I\!R^{m+n}$, le vecteur x_B est défini tel que $(x_B)_r = x_i$ où i est la i^{ieme} colonne de A dans B qui est de base à la r^{ieme} ligne. Le vecteur x_N est l'autre partie de x correspondant aux colonnes de N. Une solution primale de base $x \in I\!R^{m+n}$ associée à la base courante est telle que $x_B = B^{-1}b$ et $x_N = 0$. Si $B^{-1}b \geq 0$ alors la base B est dite réalisable. La solution de base associée est aussi réalisable. Le vecteur $y = c_B^t B^{-1}$ est la solution duale de base associée à B.

Théorème 2 *Etant donnés deux programmes linéaires duaux (P) et (D). Les propositions suivantes sont équivalentes :*

- *La valeur de la solution optimale de (P) est δ^*.*
- *Il existe une solution de base réalisable x^* de (P) et une solution de base réalisable y^* de (D) telle que $c^t x^* = y^{*t} b = \delta^*$*
- *Il existe une solution de base réalisable x^* de (P) et une solution de base réalisable y^* de (D) telle que $y^{*t}(Ax^* - b) = 0$ et $c^t x^* = \delta^*$.*

L'algorithme primal (respectivement dual) du simplexe démarre avec une base réalisable de (P) (respectivement de (D)) et itère en gardant la réalisabilité de la base tout en satisfant la troisième condition du théorème. L'optimalité est atteinte quand on obtient une solution duale (respectivement primale) de base réalisable.

Le coût réduit d'une variable x_i est égal à $c_i - c_B^t B^{-1} a_{.i}$ où $a_{.i}$ est la i^{ieme} colonne de A. Un programme linéaire est non borné (donc son dual n'est pas réalisable) quand il existe une variable x_i de coût réduit négatif et $B^{-1}a_{.i} \leq 0$. Pour plus de détails sur la programmation linéaire et l'algorithme du simplexe, le lecteur est encouragé à consulter [Chv83].

Malheureusement, une grande partie des problèmes réels ne se modélise pas sous la forme d'un programme linéaire pur. Dans plusieurs cas, les variables doivent prendre des valeurs entières et les problèmes qui en résultent sont souvent $\mathcal{NP}$-difficiles. Nous présenterons, dans la suite, des méthodes de résolution exacte de ces types de problèmes.

1.2 La méthode du "Branch and Bound"

Plusieurs problèmes d'optimisation combinatoire se formulent sous la forme d'un programme linéaire en nombres entiers (*PLNE*). Ils s'écrivent sous la forme $min\{c^t x : Ax \geq b, x \in I\!N^n\}$. Une grande partie de ces problèmes appartient à la classe des problèmes $\mathcal{NP}$-difficiles.

La méthode du "Branch and Bound" est utilisée pour résoudre d'une façon exacte des problèmes de ce type. La méthode utilise deux concepts: *le branchement* qui consiste à diviser un ensemble de solutions en sous-ensembles et *l'évaluation* qui consiste à borner ou minorer les solutions.

Dans le cas des *PLNE*, la méthode du "Branch and Bound" commence par résoudre la *relaxation linéaire* du programme linéaire, c'est à dire en enlevant les conditions d'intégralité sur les variables. Cette évaluation est toujours inférieure à la valeur de la solution optimale du *PLNE*, puisque toute solution du *PLNE* est une solution de la relaxation linéaire. Par conséquent si la solution optimale de la relaxation linéaire est entière, alors c'est une solution optimale du *PLNE*. Sinon on effectue une opération de branchement en choisissant une variable x_i^* non entière dans la solution optimale du programme linéaire que l'on a résolu. L'ensemble des solutions se divise alors en deux, celles pour lesquelles $x_i \leq \lfloor x_i^* \rfloor$ et celles pour lesquelles $x_i \geq \lfloor x_i^* \rfloor + 1$. On évalue ensuite les nouveaux nœuds et éventuellement on élague ceux qui sont inutiles. Un nœud peut être élagué dans trois cas possibles:

- Le programme linéaire n'est pas réalisable.
- La valeur de la solution est supérieure à la valeur de la meilleure solution réalisable trouvée, dans ce cas il est inutile de continuer la recherche dans

la sous-arborescence enracinée en ce nœud, puisque les bornes inférieures obtenues sont strictement croissantes suivant la profondeur de l'arbre du "Branch and Bound".

– La solution est entière, donc réalisable.

L'algorithme s'arrête quand on n'a plus de nœud à évaluer.

L'arborescence obtenue par les branchements appliqués au problème d'optimisation P est appelée arbre du "Branch and Bound", qu'on note $T = (N, E)$ où N représente l'ensemble des nœuds de l'arbre et E les arcs correspondant au processus de branchement. Il y a trois types de nœuds dans l'arbre du "Branch and Bound" pendant le déroulement de l'algorithme, le nœud courant qui est en train d'être évalué, des nœuds actifs qui sont dans la liste des nœuds qui doivent être traités, et des nœuds inactifs qui ont été élagués au cours du calcul. Quand la liste des nœuds actifs est vide, la meilleure solution réalisable obtenue est la solution optimale du problème. Notons que le choix de la stratégie de la sélection du nœud actif à traiter influe lourdement sur la taille de l'arborescence du "Branch and Bound" visitée. De même pour le choix de la variable de branchement. Ces points seront développés ultérieurement dans notre présentation.

1.3 La méthode des plans de coupes

La description sous la forme d'un programme linéaire d'un problème d'optimisation combinatoire P peut être trop grande pour être représentée explicitement en mémoire ou pour tenir dans un solveur de programmes linéaires. Dans ce cas, il suffit de résoudre une relaxation du problème d'optimisation P en enlevant la plupart de ces contraintes. L'ensemble des solutions réalisables du problème relaxé, qu'on nommera R, contient celui de P. Ainsi, pour un problème de minimisation la valeur de la fonction économique d'une solution optimale de R est inférieure ou égale à celle d'une solution optimale de P.

La méthode des plans de coupes consiste à résoudre un problème relaxé, puis à essayer d'ajouter itérativement des contraintes du problème initial, violées par la solution courante, jusqu'à ce qu'il n'y en ait plus. Pour un problème d'optimisation $min\{c^t x : Ax \geq b, x \in I\!R^n\}$, la contrainte définie par le couple (f, f_0), $f \in I\!R^n, f_0 \in I\!R$ est dite *violée* par la solution courante $\overline{x}$, si $f^t\overline{x} < f_0$ et $f^t y \geq f_0$ pour tout $y \in \{x : Ax \geq b\}$. Ces contraintes sont appelées des plans de coupes. Quand l'algorithme s'arrête, il n'y a plus de contraintes violées par la solution courante donc elle constitue une solution optimale du problème initial.

Cette méthode n'impose pas d'avoir une liste exhaustive explicite des plans de coupes, mais seulement une méthode pour générer efficacement des contraintes valides violées. L'identification de ces contraintes est appelée le *problème de séparation.*

Le problème de séparation:
Etant donnés un ensemble de contraintes d'un programme linéaire et un vecteur $x \in \mathbb{R}^n$, il faut ou bien prouver que x satisfait toutes les contraintes ou bien trouver une inéquation qui est violée par x.

L'algorithme des plans de coupes est décrit dans Algorithme 1. Il s'arrête quand il n'y a plus de contrainte violée dans le système $Ax \geq b$. La solution du programme linéaire est optimale dans ce cas. Malheureusement dans le cas des programmes linéaires en nombres entiers cette solution peut être fractionnaire bien qu'il n'y ait plus d'inéquation violée dans le système des contraintes $Ax \geq b$. Comment peut-on obtenir des plans de coupes pour des problèmes en nombres entiers?

Pour ce faire, on va revenir à la première formulation d'un problème d'optimisation combinatoire. En effet, on a vu au début de ce chapitre qu'un problème peut s'écrire sous la forme de :

$$min\{c^t x : x \in conv\{x^S : S \in \mathcal{S}\}\}$$

où x^S est le vecteur d'incidence d'une solution S du problème et $\mathcal{S}$ définit l'ensemble des solutions du problème.

L'enveloppe convexe de l'ensemble des vecteurs d'incidence des solutions du problème en nombres entiers définit en général un polyèdre borné.

$$P = conv\{x^S : S \in \mathcal{S}\}$$

Avant de continuer notre exposé, nous avons besoin de rappeler quelques éléments de la théorie polyédrale. Pour une compréhension complète de ce domaine nous vous renvoyons vers [Chv83].

Un polyèdre P est un ensemble de points défini par un ensemble fini d'équations et d'inéquations linéaires, il s'écrit sous la forme $P = \{x \in \mathbb{R}^n : Ax \geq b\}$, où $A \in \mathbb{R}^{m \times n}$ et $b \in \mathbb{R}^m$. Un polyèdre est dit borné si le $max_{x \in P} \|x\| < \infty$ où $\|.\|$ est une norme de $\mathbb{R}^n$.

Algorithme 1 L'algorithme des plans de coupes

/* On veut résoudre le problème d'optimisation combinatoire $min\{c^t x : Ax \geq b, x \in I\!R^n\}$ avec $A \in I\!R^{m \times n}$ et $b \in I\!R^m$ */

Initialiser le programme linéaire par le sous système de contraintes (A_1, b_1) avec $A_1 \in I\!R^{m_1 \times n}$ et $b_1 \in I\!R_1^m$ avec $m_1 << m$;
Calculer la solution optimale $\overline{x}$ du programme linéaire $c^t\overline{x} = min\{c^t x : A_1 x \geq b_1, x \in I\!R^n\}$;

Tant que *Nombre-de-contrainte = rechercher-des-contraintes-violées()* **faire**
/* On recherche les contraintes violées par $\overline{x}$ dans le système $Ax \geq b$, le nombre de contraintes violées est retourné dans la variables *Nombre-contrainte* */
Ajouter l'ensemble des contraintes violées défini par (A_2, b_2) au programme linéaire;
$A_1 = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$ et $b_1 = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$;
Calculer la solution optimale $\overline{x}$ du programme linéaire $c^t\overline{x} = min\{c^t x : A_1 x \geq b_1, x \in I\!R^n\}$;
Fin Tant que

$x^* = \overline{x}$ est la solution optimale de $min\{c^t x : Ax \geq b, x \in I\!R^n\}$;

Un polytope est un polyèdre borné. Il peut être défini comme l'enveloppe convexe d'un ensemble fini de points.

La dimension d'un polyèdre P, notée $dim(P)$ est la dimension du plus petit espace affine contenant P. Il est dit de pleine dimension si $dim(P) = n$.

Une inéquation $fx \geq f_0$ est dite valide si est seulement si $f\overline{x} \geq f_0$ pour tout $\overline{x} \in \{x \in I\!R^n : Ax \geq b\}$. Si l'intersection entre P et $\{x \in I\!R^n : fx = f_0\}$ est non vide et différente de P alors $F = \{x \in P : fx = f_0\}$ est dite *face* propre de P définie par l'inéquation valide $fx \geq f_0$. Les faces propres F de dimension maximale c'est à dire tel que $dim(F) = dim(P) - 1$ sont appelées *facettes* de P.

Tout polyèdre P peut être écrit sous la forme d'un système formé d'inéquations linéaires. De plus, s'il est de pleine dimension alors il admet une description unique par des inéquations linéaires valides représentant des facettes, à une multiplication près de ses deux membres par un nombre positif. Dans le cas où P ne serait pas de pleine dimension, le polyèdre est défini par ses facettes, plus un système maximal indépendant d'équations.

Pour résoudre un programme linéaire en nombres entiers par la méthode des plans de coupes, on peut tout d'abord utiliser des plans généraux de coupes de l'enveloppe convexe des solutions du problème. Ces coupes ne sont pas spécifiques au problème résolu, et peuvent être utilisées pour tous les problèmes linéaires en nombres entiers. Les plans de coupe de Gomory [Gom58] ou les coupes générées par la méthode dite du "lift and project" introduite par Balas, Céria et Cornuéjols [BCC93a, BCC93b, BCC94] en sont un exemple.

L'utilisation de ces plans s'avère souvent limitée pendant la résolution de problèmes difficile. D'où l'intérêt de l'étude des plans de coupes spécifiques au problème résolu.

L'utilisation des facettes de l'enveloppe convexe de l'ensemble des solutions d'un problème linéaire en nombres entiers comme plans de coupes donne des résultats très satisfaisants. En effet, chaque facette du polytope induit une inégalité valide. Les facettes d'un polytope sont les meilleurs plans de coupe qu'on puisse générer. Pendant la procédure de séparation on utilise soit des algorithmes exacts de séparation pour générer les facettes connues, soit des heuristiques de séparation qui ne garantissent malheureusement pas l'absence d'inégalités violées quand ils n'en trouvent pas.

Grötschel, Lovàsz et Schrijver [GLS88] et Padberg et Rao[PR82] ont démontré 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.

Pour des problèmes linéaires en nombres entiers appartenant à la classe des problèmes $\mathcal{NP}$-difficiles, on ne connaît pas la description linéaire complète du

polytope associé. Et même si on possède une description partielle des facettes du polytope du problème étudié, on ne connaît pas d’algorithmes exacts de séparation pour toutes ces familles de facettes. Ainsi, souvent la méthode des plans de coupes s’arrête avant d’avoir trouvé la solution optimale du problème. Notons qu’on parlera dorénavant de méthode de coupes polyédrales quand on n’utilise que les facettes du polytope comme plans de coupes.

1.4 La méthode du “Branch and Cut”

On vient de voir qu’on ne peut souvent pas résoudre efficacement des problèmes $\mathcal{NP}$-difficiles par la méthode des coupes polyédrales. De même, bien que l’algorithme du “Branch and Bound” puisse être très performant pour une certaine classe de problèmes, par exemple le problème de sac à dos, il reste très limité quand il est difficile de calculer une borne inférieure du problème d’optimisation. C’est le cas des problèmes possèdant un nombre exponentiel d’inéquations dans leur formulation en un programme linéaire en nombres entiers.

L’algorithme du “Branch and Cut” est une méthode qui conjugue les efforts de l’algorithme du “Branch and Bound” et de la méthode des coupes polyédrales. Ainsi, pour résoudre un programme linéaire en nombres entiers, le “Branch and Cut” commence par résoudre une relaxation du problème puis il applique la méthode des coupes polyédrales sur la solution trouvée. Si celle-ci n’arrive pas a obtenir une solution entière alors le problème est divisé en plusieurs sous-problèmes qui seront résolus de la même façon. L’algorithme de base est décrit dans algorithme 2.

Une telle approche a été utilisée pour la première fois pour le problème de “linear ordering” par Grötschel, Junger et Reinelt [GJR84]. Le terme de “Branch and Cut” était introduit par Padberg et Rinaldi [PR87, PR91] pour un algorithme de résolution du problème du voyageur de commerce. Dans [PR91], Padberg et Rinaldi ont établi le premier état de l’art de cet algorithme, en introduisant des nouvelles procédures telles que la génération de colonnes, des procédures de séparation sophistiquées et une utilisation efficace du solveur du programme linéaire.

Une présentation détaillée de cet algorithme fera l’objet du chapitre 3.

Algorithme 2 L'algorithme du "Branch and Cut"

/* On veut résoudre le problème d'optimisation combinatoire $min\{c^t x : Ax \geq b, x \in I\!R^n\}$ avec $A \in I\!R^{m \times n}$ et $b \in I\!R^m$ */

Liste-des-problèmes = vide;
Initialiser le programme linéaire par le sous système de contraintes (A_1, b_1) avec $A_1 \in I\!R^{m_1 \times n}$ et $b_1 \in I\!R_1^m$ avec $m_1 << m$;

/* Etapes d'évaluation d'un problème */
Calculer la solution optimale $\overline{x}$ du programme linéaire $c^t\overline{x} = min\{c^t x : A_1 x \geq b_1, x \in I\!R^n\}$;
Solution-courante = Appliquer-la-méthode-des-coupes-polyédrales();
/* Fin étapes d'évaluation */

Si *Solution courante est réalisable* **alors**
  $x^* = \overline{x}$ est la solution optimale de $min\{c^t x : Ax \geq b, x \in I\!R^n\}$;
Sinon
  Ajouter le problème dans *Liste-des-problèmes*;
Fin Si

Tant que *Liste-des-problèmes non vide* **faire**
  Sélectionner un problème;
  Brancher le problème en deux ou plusieurs sous problèmes;
  Résoudre les nouveaux sous problèmes (en appliquant les étapes d'évaluation);
Fin Tant que

1.5 Le pricing ou la génération de colonnes

Quand un programme linéaire admet un grand nombre de variables, il est très difficile de le résoudre explicitement. Dans ce cas, on résout le programme linéaire sur un sous-ensemble de variables et on vérifie ensuite si l'ajout d'une variable qui n'appartient pas au programme linéaire courant peut améliorer la solution optimale. En effet pour un programme linéaire du type $min\{c^t x : Ax \leq b, x \geq 0\}$, une variable dont le coût réduit est négatif, peut améliorer la valeur de la solution optimale. Rappelons ici, que le coût réduit r_j d'une variable j hors base est égal à $c_j - y^t a_{.j}$ où $a_{.j} \in I\!R^m$ est la colonne associée à la variable j, c_j est le coefficient dans la fonction économique correspondant à une solution de base du programme linéaire et $y \in I\!R^m$ la solution du dual. Le calcul du coût réduit d'une variable est appelé "pricing".

Comme dans la méthode des coupes polyédrales, il est inutile d'avoir la liste explicite des variables pour l'algorithme de génération de colonnes. Il suffit d'avoir une méthode permettant de générer les variables du problème initial qui ont un coût réduit négatif. Ce problème porte le nom du problème du "pricing":

Le problème de "pricing":
Etant donnée une classe de variables d'un problème linéaire et la valeur des variables duales de la solution de base. Il faut soit prouver que toutes les variables de cette classe ont un coût réduit positif, soit trouver une variable hors base dont le coût réduit est négatif.

L'algorithme de génération de colonnes est décrit dans Algorithme 3. Il peut être intégré dans la méthode du "Branch and Cut" pour pouvoir résoudre des problèmes d'optimisation difficiles avec un grand nombre de variables et de contraintes. La méthode obtenue porte le nom du " Branch and Cut and Price ". C'est l'algorithme qu'on va utiliser pour résoudre le problème du voyageur de commerce. Mais pour ne pas alourdir notre texte, nous appellerons désormais cette méthode, l'algorithme du "Branch and Cut".

Algorithme 3 L'algorithme de génération de colonnes

/* On veut résoudre le problème d'optimisation combinatoire $min\{c^t x : Ax \geq b, x \in I\!R^n\}$ avec $A \in I\!R^{m \times n}$ et $b \in I\!R^m$ */

continue-le-pricing = vrai;
Sélectionner un petit sous-ensemble J des variables $\{1..m\}$;

Tant que *continue-le-pricing* **faire**
  Calculer la solution de base $\overline{x}_J$ du programme linéaire $c_J^t \overline{x}_J = min\{c_J^t x_J : A_J x_J \geq b, x_J \geq 0, x_J \in I\!R^{|j|}\}$ et déterminer les valeurs des variables duales $\overline{y}$;
  Pour *i qui n'est pas dans J* ***faire***
    Si $r_i = c_i - y^t a_{.i} \geq 0$ ***alors***
      continue-le-pricing = faux;
    Sinon
      Ajouter la colonne j avec le $r_j < 0$ *à J;*
      continue-le-pricing = vrai;
      Sortir de la boucle **Pour***;*
    Fin Si
  Fin Pour
Fin Tant que

$\overline{x}_J$ est la solution optimale de $min\{c^t x : Ax \geq b, x \in I\!R^n\}$;

Chapitre 2

Approche polyédrale du problème du voyageur de commerce

Ce chapitre est consacré à une approche polyédrale de l'étude du problème du voyageur de commerce. Nous commencerons par une modélisation du problème sous la forme d'un programme linéaire en nombres entiers. Nous définirons ensuite le polytope du problème du voyageur de commerce et nous donnerons ses principales propriétés. Nous décrirons enfin quelques facettes connues du polytope du problème du voyageur de commerce.

2.1 Le problème du voyageur de commerce

Avant de donner la formulation du problème du voyageur de commerce en programme linéaire en nombres entiers, rappelons quelques définitions, qui nous serviront tout au long de cette thèse.

Un graphe $G = (V, E)$ est défini par l'ensemble de ses sommets V, et l'ensemble de ses arêtes E. On note $K_n = (V_n, E_n)$ le graphe non orienté complet sur n sommets.

Soit x une fonction coût de $E \rightarrow I\!R^{|E|}$ qui associe à chaque arête $e \in E$ un coût x_e. Pour tout $F \subseteq E_n$, on note par $x(F)$ la somme $\sum_{e \in F} x_e$.

Pour tout $S \subseteq V_n$, on note par $\gamma(S) = \{uv \in E_n : u, v \in S\}$ l'ensemble des arêtes de E_n dont les deux extrémités sont dans S, et par $\delta(S) = \{uv \in E_n : u \in S, v \in V_n - S\}$ l'ensemble des arêtes possédant exactement une seule extrémité dans S. Par abus d'écriture on notera $\delta(\{v\}) = \delta(v)$, pour tout $v \in V_n$.

Pour tout $S \subset V_n$ et $T \subset V_n \backslash S$, on note par $(S : T) = (T : S)$ l'ensemble des arêtes dont une extrémité est dans S et l'autre dans T.

Pour un graphe $G = (V, E)$, on note $G(S)$ le sous graphe induit par S. Et à un vecteur $x \in I\!R^{|E|}$ on associe un graphe $G_x = (V, E, x)$ dont l'ensemble des arêtes contient toutes les arêtes de E qui ont une composante non nulle dans x. Le poids d'une arête e de G_x est égal à x_e.

Soient $K_n = (V_n, E_n)$ le graphe non orienté complet sur n sommets, et $c \in I\!R^{|E_n|}$ une fonction qui associe à chaque arête de K_n une longueur c_e. Le problème de voyageur de commerce consiste à trouver un cycle hamiltonien H^* tel que $c(H^*) = min\{c(H) : H \in \mathcal{H}_n\}$, où $\mathcal{H}_n$ représente l'ensemble des cycles hamiltoniens de K_n.

Un cycle Hamiltonien est un sous graphe $H = (V_n, E)$ de K_n satisfaisant:

$$\text{(a)} \quad \text{chaque sommet de } H \text{ est de degré 2.} \tag{2.1.2.1}$$

$$\text{(b)} \quad H \text{ est connexe.} \tag{2.1.2.2}$$

On associe à tout $H \in \mathcal{H}_n$ un vecteur d'incidence unique $x \in \{0, 1\}^{|E_n|}$ défini par:

$$x_e^H = \begin{cases} 1 & \text{si } e \in E \\ 0 & \text{sinon} \end{cases}$$

Le problème du voyageur de commerce s'écrit sous la forme du programme linéaire en nombres entiers suivant:

$$\begin{array}{lrll} \text{minimiser} & c^T x & & \\ \text{sous} & x(\delta(v)) = 2 & \text{pour tout } v \in V_n & (2.1.2.3) \\ & x(\delta(S)) \geq 2 & \text{pour tout } S \subset V_n, S \neq V_n, S \neq \emptyset & (2.1.2.4) \\ & 0 \leq x_e \leq 1 & \text{pour tout } e \in E_n & (2.1.2.5) \\ & x_e \in \{0, 1\} & \text{pour tout } e \in E_n & (2.1.2.6) \end{array}$$

L'équation (2.1.2.3) signifie que chaque sommet est exactement incident à 2 arêtes, elle peut être écrite sous la forme suivante:

$$A_n x = \mathbf{2} \tag{2.1.2.7}$$

où A_n est la matrice d'incidence de K_n et $\mathbf{2}$ est un vecteur de $I\!R^n$ dont les composantes sont égales à 2. Ces inéquations portent le nom d'*équations de degré*.

Les *inéquations de sous-tour* (2.1.2.4) assurent que H est connexe, puisque pour tout ensemble $S \subset V_n, S \neq V_n, S \neq \emptyset$, le cocycle $\delta(S)$ intersecte au moins 2 fois un cycle hamiltonien. Ces inéquations peuvent s'écrire sous une forme équivalente :

$$x(\gamma(S)) \leq |S| - 1 \quad \text{pour tout } S \subset V_n, S \neq V_n, S \neq \emptyset \tag{2.1.2.8}$$

Les instances qui peuvent être résolues par la méthode du "Branch and Bound" avec évalution par un programme linéaire restent de petite taille. La raison est la différence trop importante entre la valeur de la solution optimale de la relaxation linéaire et celle du *PLNE*. Pour combler cette différence, il faut renforcer la description linéaire du problème. D'où la nécessité de l'étude du polytope de ce problème pour pouvoir utiliser des méthodes plus sophistiquées telle que le "Branch and Cut". Nous définirons dans la suite le polytope du problème du voyageur de commerce et nous décrirons quelques unes de ses facettes.

2.2 Le polytope du problème du voyageur de commerce

Le polytope du voyageur de commerce symétrique $STSP(n)$ est l'enveloppe convexe de l'ensemble des vecteurs d'incidence des cycles Hamiltoniens de K_n.

$$STSP(n) = conv\{x^H : H \in \mathcal{H}_n\}$$

Les vecteurs d'incidences des cycles hamiltoniens forment les sommets du polytope $STSP(n)$. D'après un résultat classique de Farkas, Weyl et Minkowski [Sch86], $STSP(n)$ peut être décrit par un système d'équations et d'inéquations linéaires, i.e. $STSP(n)= \{x : Ax \geq b\}$, où chaque inéquation induit une facette de $STSP(n)$. Malheureusement on ne connaît qu'une petite partie de la description linéaire du problème du voyageur de commerce, puisque aucune description linéaire complète ne peut être donnée à un problème $\mathcal{NP}$-difficile, à moins que $\mathcal{NP}$ = co-$\mathcal{NP}$ (voir Karp et Papadimitriou [KP82]). Cependant une connaissance partielle de cette description permet souvent de trouver la solution optimale du problème en utilisant la méthode du "Branch and Cut". La description partielle du polytope $STSP(n)$ fera l'objet de paragraphe suivant.

2.3 Quelques facettes de $STSP(n)$

Le polytope du voyageur de commerce $STSP(n)$ peut s'écrire sous la forme d'un système d'inéquations $Ax \geq b$ représentant les facettes du polyèdre, et un système minimal d'équations $A^{=}x = b^{=}$ formé par les équations de degré. Le polytope $STSP(n)$ n'est pas de pleine dimension puisqu'il satisfait les contraintes de degré. En effet, $dim(STSP(n)) = m - n$ avec $m = |E_n| = \frac{n(n-1)}{2}$. Donc les facettes qui forment le polyèdre du voyageur de commerce ne sont pas définies d'une manière unique à une multiplication près par un scalaire.

Théorème 3 *Soit P un polyèdre qui n'est pas de pleine dimension, et soit F une facette de P, et $cx \geq c_0$ une inégalité représentant F. L'inéquation $fx \geq f_0$ représente la même facette F si et seulement si*

$$\exists \alpha \in I\!R^{*}_{+} \text{ et } \lambda \in I\!R^{p} \text{ où } p = |A^{=}| \text{ et tel que } (f, f_0) = (\alpha c + \lambda A^{=}, \alpha c_0 + \lambda b^{=})$$

Ainsi d'après le théorème précédent, si $hx \geq h_0$ est une inégalité représentant une facette du $STSP(n)$ alors $fx \geq f_0$ tel que $f = \lambda A_n + \Pi h$, $f_0 = \lambda \mathbf{2} + \Pi h_0$ avec $\Pi > 0$ et $\lambda \in I\!R^{n}$ définit la même facette. On dit que les deux inéquations sont équivalentes.

En conséquence les inéquations représentant des facettes de $STSP(n)$ sont décrites sous plusieurs formes dans la littérature. Les facettes que nous présenterons dans ce document sont décrites sous la forme suivante.

Définition 1 *Soit $\mathcal{S} = \{S_1, \ldots, S_p\}$ une collection de sous ensemble de V_n. Une inéquation s'écrit sous une forme cocyclique si elle est écrite comme suit :*

$$\sum_{i=1}^{p} \alpha_i x(\delta(S_i)) \geq r(\mathcal{S})$$

où α_i est un entier qui dépend de $S_i \in \mathcal{S}$ et $r(\mathcal{S})$ est une fonction de

$$min\{\sum_{i=1}^{p} \alpha_i |H \cap S_i|, \text{ pour tout } H \in \mathcal{H}\}$$

Malheureusement, toutes les facettes du $STSP(n)$ ne peuvent pas être écrites sous une forme cocyclique, comme les inégalités hypohamiltoniennes [CFN85] par exemple.

Définition 2 *On dit qu'une inégalité $fx \geq f_0$ est triangulaire serrée ou TS si:*

– *les coefficients f_e satisfont l'inégalité triangulaire c.à.d.*

$$\forall u \neq v \neq w \quad f(u,v) + f(u,w) \geq f(v,w)$$

$$- \forall u \in V_n \quad \exists v \neq w \in V_n - \{u\} \quad / \quad f(u,v) + f(u,w) = f(v,w)$$

Théorème 4 *Soit $hx \geq h_0$ une inégalité définie sur $\mathbb{R}^{|E_n|}$, on obtient une inéquation $fx \geq f_0$ triangulaire serrée équivalente à $hx \geq h_0$, si on pose:*

$$f = \lambda A_n + \Pi h \text{ et } f_0 = \lambda \mathbf{2} + \Pi h_0 \text{ avec } \Pi > 0$$

et

$$\lambda_u = \frac{\Pi}{2} max\{h(v,w) - h(u,v) - h(u,w) / v, w \in V_n - \{u\}, v \neq w\}$$

D. Naddef et G. Rinaldi [NR91, NR92, NR93] ont utilisé la forme triangulaire serrée pour expliquer la relation entre $STSP(n)$ et $GTSP(n)$ le polytope représentant l'enveloppe convexe des tours du graphe G.

Théorème 5 *(Naddef et Rinaldi)*
Toute facette de $STSP(n)$ en forme TS est une facette de $GTSP(n)$.

Dans la suite nous présenterons quelques facettes connues du problème du voyageur de commerce.

2.3.1 Inégalités triviales et du sous-tour

Notons tout d'abord que les inégalités du type $x_e \geq 0, \forall e \in E$ sont des facettes triviales du $STSP(n)$ [GP79].

Grötschel et Padberg [GP79] ont prouvé que les inégalités du sous-tour $x(\delta(S)) \geq 2$, introduites par Dantzig, Fulkerson et Johnson [DFJ54], sont des facettes du $STSP(n)$. Leur validité est triviale. En effet, un cycle hamiltonien a au moins deux arêtes en commun avec un cocycle associé à S, pour tout $S \subset V_n, S \neq \emptyset$.

2.3.2 Inégalités de peignes

Les inégalités de *peignes* (*Comb Inequalities*) sont définies par la famille $\mathcal{S} = \{H, T_1, \ldots, T_{2k+1}\}$ de sous ensemble de V_n tel que $\alpha_s = 1$ pour tout $S \in \mathcal{S}$ et $r(\mathcal{S}) = 6k + 4$.

H est appelé le manche ou la poignée et les T_i sont appelés des dents. Ils vérifient les propriétés suivantes:

$$\begin{array}{lll} (i) & \forall i = 1, ..., 2k+1 & H \cap T_i \neq \emptyset \\ (ii) & \forall i = 1, ..., 2k+1 & T_i \setminus H \neq \emptyset \\ (iii) & \forall 1 \leq i < j \leq 2k+1 & T_i \cap T_j = \emptyset \end{array}$$

L'inégalité s'écrit:

$$x(\delta(H)) + \sum_{i=1}^{2k+1} x(\delta(T_i)) \geq 6k + 4$$

Il existe des classes particulières d'inégalités de peignes. Ainsi, si on remplace la propriété (i) par:

$$(i') \qquad \forall i = 1, ..., 2k+1 \quad |H \cap T_i| = 1$$

alors les inégalités portent le nom de *Chvátal comb* [Chv73]. Si toutes les dents sont de cardinalité 2, alors les inégalités se nomment inégalité des *2-couplages*.

Grötschel et Padberg [GP79] ont démontré que les inégalités de peignes forment des facettes du $STSP(n)$ quand $n \geq 6$.

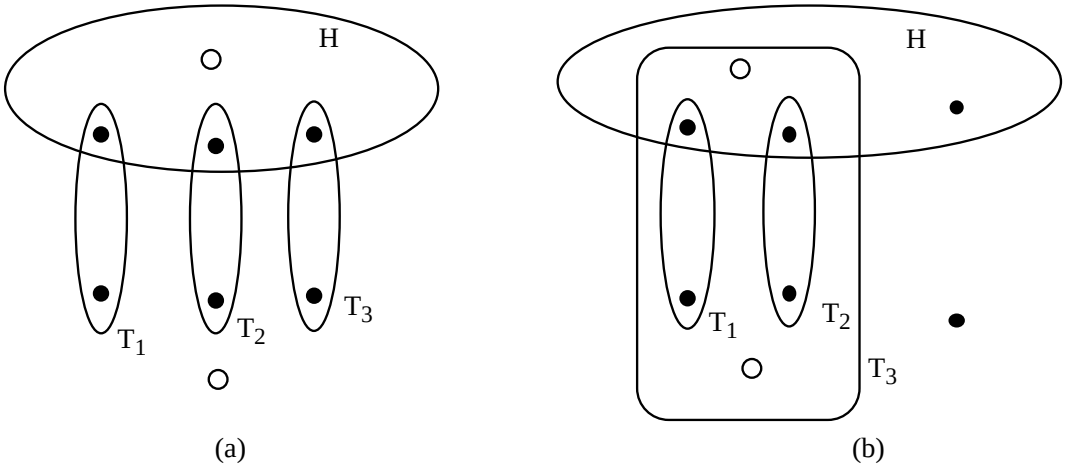

FIG. 2.1 – *Exemple de peigne*

La figure 2.1 montre un exemple de peigne avec $k = 1$. Notons qu'on utilisera dans les figures représentant des facettes de $STSP(n)$ les conventions suivantes: les ensembles possédant un point noir doivent être non vides, ceux possèdant un point blanc peuvent contenir des sommets et ceux ne possèdant aucun point doivent être vide. Un point ne représente pas un sommet unique, mais un ensemble de sommets pouvant se trouver dans cette région.

Notons que si on remplace l'un ou plusieurs des ensembles qui forment le peigne par leurs complémentaires, alors on obtient la même inégalité (voir figure 2.1 (b)). En effet, le cocycle engendré par un ensemble de sommets est égal au cocycle engendré par son complémentaire, d'où l'inégalité associée ne change pas.

Rapporter les preuves de validité des inégalités de peigne n'est pas l'objet de cette présentation. On se contentera de donner une explication intuitive de cette validité inspirée de l'article de Naddef et Pochet [NP98]. Cette explication peut permettre de comprendre pourquoi la famille $\mathcal{S}$ qui définit les inégalités sous forme cocyclique est partitionnée en deux sous ensembles formés d'un manche et de plusieurs dents.

Supposons que k=1. Si le cocycle engendré par une dent T_i intersecte un cycle hamiltonien Γ en exactement deux arêtes, alors le cocycle du manche intersecte Γ en au moins trois arêtes, voir figure 2.2 (a). Or sachant qu'un cocycle non vide intersecte un cycle hamiltonien en un nombre pair d'arêtes, alors $|\Gamma \cap \delta(H)| \geq 4$. D'où l'inégalité de peigne est satisfaite dans ce cas par tout vecteur x représentant un tel cycle hamiltonien. Supposons maintenant qu'il existe une dent $T_{i^\star}$ telle que $|\Gamma \cap \delta(T_{i^\star})| \neq 2$ alors en appliquant le même principe, on a $|\Gamma \cap \delta(T_{i^\star})| \geq 4$ et le manche et les 2 autres dents ont au moins 2 arêtes de leur cocycle dans Γ, voir figure 2.2 (b). D'où, de nouveau, l'inégalité de peigne est satisfaite par tous les vecteurs x représentant un cycle hamiltonien.

La preuve de la validité dans le cas général peut être construite par induction sur le nombre des dents. Supposons que les inégalités sont valides jusqu'à un certain ordre $k - 1 \geq 1$, et démontrons que ceci est vrai pour k. Soit un cycle hamiltonien Γ. Si $|\Gamma \cap \delta(T_i)| = 2$ pour i=1, ..., 2k+1, alors $|\Gamma \cap \delta(H)| \geq 2k + 1$. Or puisqu'un cocycle non vide intersecte un cycle hamiltonien en un nombre pair d'arêtes alors $|\Gamma \cap \delta(H)| \geq 2k+2$, d'où la validité pour tout vecteur x représentant un tel cycle hamiltonien. Supposons maintenant que $|\Gamma \cap \delta(T_{i^\star})| \geq 4$, pour un $i^\star$ donné, et par commodité supposons que $i^\star = 2k + 1$. Par hypothèse de récurrence on a $|\Gamma \cap \delta(H)| + \sum_{i=1}^{2k-1} |T_i| \geq 6k + 4 - 6$ en ajoutant $|\Gamma \cap \delta(T_{2k+1})| \geq 4$ et $|\Gamma \cap \delta(T_{2k})| \geq 2$, on prouve la validité des inégalités des peignes. $\diamondsuit$

Le même type d'argument peut être utilisé pour démonter la validité de la

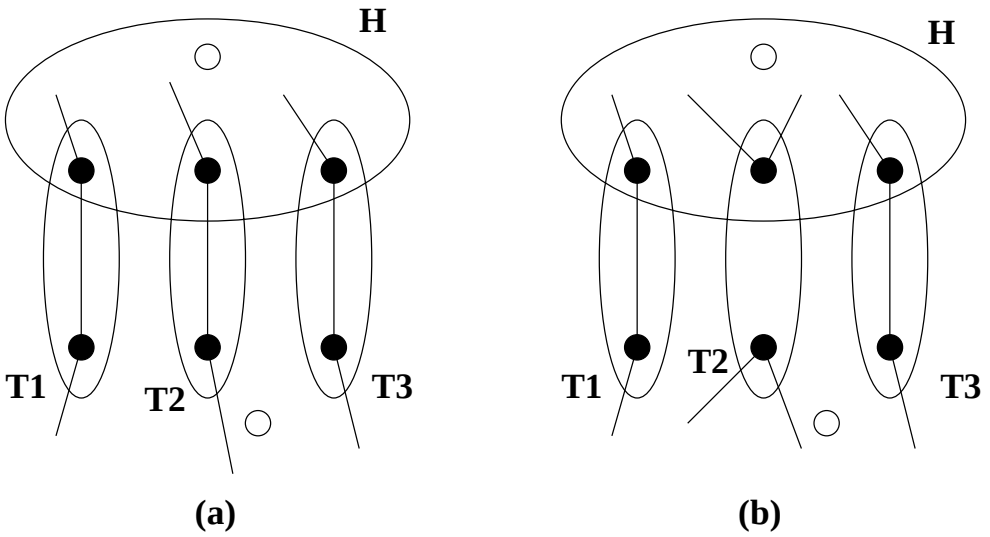

FIG. 2.2 – *Intersection d'un peigne avec un cycle hamiltonien*

plupart des inégalités définissant des facettes du $STSP(n)$ présentées ci-dessous.

2.3.3 Inégalités des chemins, d'étoiles et bi-emboitées

Les inégalités d'*étoiles* (*Star inequalities*) ont été introduites par Fleischman [Fle88]. Cet ensemble d'inégalités contient l'ensemble des inégalités des *chemins* (*Path inequalities*) définies par Cornuéjols, Fonlupt et Naddef [CFN85], et il est contenu dans l'ensemble des inégalités *bi-emboitées* (*Binested inequalities*) introduites par Naddef [Nad92]. On se limitera dans la suite à la présentation des inégalités d'étoiles.

Les inégalités d'étoiles sont définies par la famille $\mathcal{S} = \{H_1, \ldots, H_h, T_1, \ldots, T_t\}$, où t est un nombre impair. Ces ensembles doivent satisfaire les contraintes suivantes:

$$\begin{array}{rl} H_1 \subset H_2 \subset \cdots \subset H_h & \\ H_1 \cap T_j \neq \emptyset & \text{pour } j = 1, \ldots, t \\ T_j \setminus H_h \neq \emptyset & \text{pour } j = 1, \ldots, t \\ (H_{i+1} \setminus H_i) \setminus \bigcup_{j=1}^{t} T_j = \emptyset & \forall i, 1 \leq i \leq h-1 \end{array}$$

L'inégalité d'étoile s'écrit:

$$\sum_{i=1}^{p} \alpha_i x(\delta(H_i)) + \sum_{j=1}^{t} \beta_j x(\delta(T_j)) \geq (t+1) \sum_{i=1}^{p} \alpha_i + 2 \sum_{j=1}^{t} \beta_j$$

où les α_i et les β_j sont des entiers qui vérifient la condition suivante:

Si on définit l'intervalle relatif à une dent T_j comme l'ensemble maximal des indices des manches qui ont une intersection avec T_j, et si pour un intervalle $I = \{\ell, \ell+1, ..., \ell+r\}$, on appelle le poids de l'intervalle $\sum_{i=\ell}^{\ell+r} \alpha_i$; alors pour chaque dent T_j, on a $\beta_j \geq$ poids maximum d'un intervalle relatif à T_j.

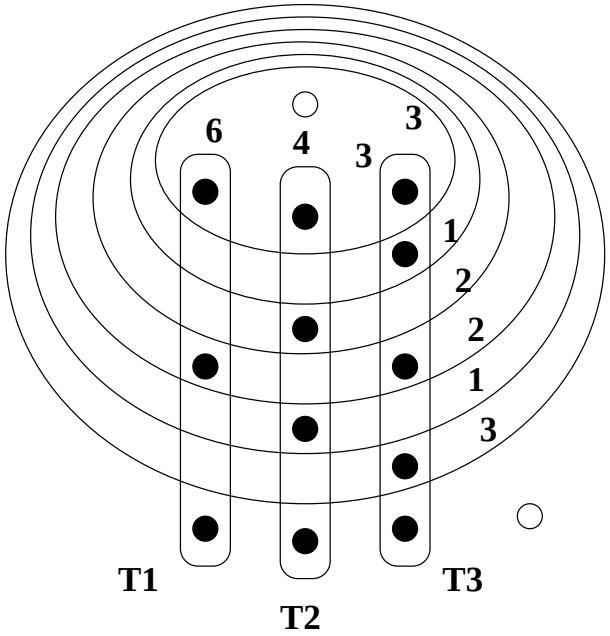

FIG. 2.3 – *Exemple de chemin ($h = 6$, $t = 3$)*

Quand tous les intervalles relatifs à une dent T_j ont les mêmes poids, et quand les β_j sont exactement égaux à cette valeur, les inégalités sont appelées inégalités de chemins. La figure 2.3 montre un exemple d'inégalité de chemin avec $h = 6$ et $t = 3$.

2.3.4 Inégalités d'échelles

Les inégalités d'*échelles* (*Ladder inequalities*) sont définies par la famille

$$\mathcal{S} = \{H_1, H_2, P_1, P_2, T_1, \ldots, T_t, D_1, \ldots, D_m\}$$

avec $t+m$ est pair et au moins égal à 2. Les ensembles H_i, P_i, T_i et D_i sont appelés respectivement *manches, dents pendantes, dents régulières et dents dégénérées*. Les inégalités associées à $\mathcal{S}$ est:

$$\sum_{i=1}^{2} x(\delta(H_i)) + \sum_{i=1}^{2} x(\delta(P_i)) + \sum_{j=1}^{t} x(\delta(T_j)) + \sum_{j=1}^{m} 2x(\delta(D_j)) - 2x(H_1 \cap P_1 : H_2 \cap P_2) \geq 2(2t + 3m + 4)$$

Remarquer que cette inégalité n'est pas écrite sous forme cocyclique à cause du terme $x(H_1 \cap P_1 : H_2 \cap P_2)$. Les inégalités d'*échelle* définissent des facettes de $STSP(n)$, ($n \geq 8$) si :

$$\begin{aligned}
H_1 \cap H_2 &= \emptyset \\
H_1 \cap P_1 \neq \emptyset, H_2 \cap P_1 &= \emptyset \\
H_2 \cap P_2 \neq \emptyset, H_1 \cap P_2 &= \emptyset \\
P_1 \setminus H_1 \neq \emptyset, P_2 \setminus H_2 &\neq \emptyset \\
T_j \cap H_i \neq \emptyset \qquad & \text{pour } i = 1, 2, \ \text{ pour } j \geq 0 \\
D_j \cap H_i \neq \emptyset \qquad & \text{pour } i = 1, 2, \ \text{ pour } j \geq 0
\end{aligned}$$

Ces inégalités ont été introduites par Boyd et Cunningham [BC91]. La preuve qu'elles forment des facettes de $STSP(n)$ a été donnée par Boyd, Cunningham, Queyranne et Wang [BCQW93]. Un exemple d'échelle est montré dans la figure 2.4.

La même inégalité est obtenue en remplaçant H_2 par son complément sur V_n, voir figure 2.4. Dans ce cas les conditions que doivent vérifier les S_i sont:

$$\begin{aligned}
H_1 \subset H_2 & \\
H_1 \cap P_1 \neq \emptyset, \ H_2 \supset P_1 & \\
H_2 \cap P_2 \neq \emptyset, \ H_1 \cap P_2 = \emptyset & \\
P_1 \setminus H_1 \neq \emptyset, \ P_2 \setminus H_2 \neq \emptyset & \\
T_j \cap H_1 \neq \emptyset, \ T_j \setminus H_2 \neq \emptyset \quad & \text{pour } j \geq 0 \\
D_j \cap H_1 \neq \emptyset, \ D_j \setminus H_2 \neq \emptyset \quad & \text{pour } j \geq 0
\end{aligned}$$

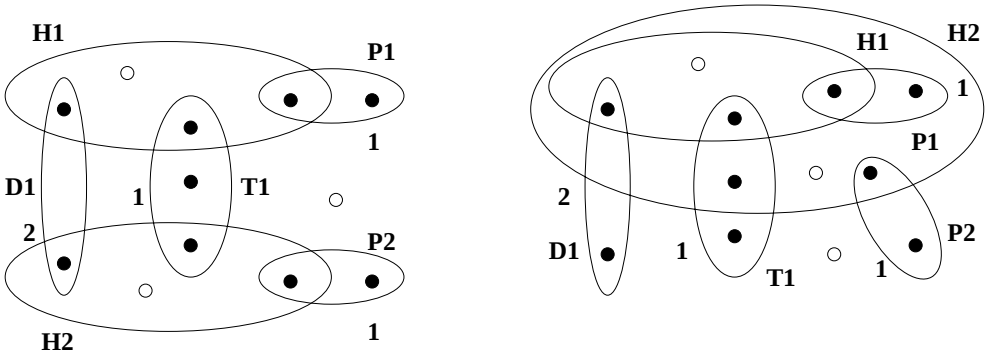

FIG. 2.4 – *Exemple d'échelle*

L'inégalité s'écrit alors sous la forme suivante:

$$\sum_{i=1}^{2} x(\delta(H_i)) + \sum_{i=1}^{2} x(\delta(P_i)) + \sum_{j=1}^{t} x(\delta(T_j)) + \sum_{j=1}^{m} 2x(\delta(D_j)) - 2x(P_1 \cap H_1 : P_2 \setminus H_2) \geq 2(2t + 3m + 4)$$

2.3.5 Inégalités de bipartition

Les inégalités de *bipartition* (*Bipartition inequalities*) sont définies par le squelette $\mathcal{S} = \{H_1, \ldots, H_h, T_1, \ldots, T_t\}$ et les coefficients β_j associés aux dents T_j.

Ces éléments vérifient les propriétés suivantes:

$H_i \cap H_j = \emptyset$	pour $1 \leq i < j \leq h$
$T_i \cap T_j = \emptyset$	pour $1 \leq i < j \leq t$
$T_j \setminus H_i \neq \emptyset$	pour $1 \leq i \leq h$ et $1 \leq j \leq t$
$t_j = \mid \{i : T_j \cap H_i \neq \emptyset\} \mid \geq 1$	pour $1 \leq j \leq t$
$h_i = \mid \{j : H_i \cap T_j \neq \emptyset\} \mid \geq 3$ et impaire	pour $1 \leq i \leq h$
$\beta_j = 1$ si $T_j \setminus (\bigcup_{i=1}^{h} H_i) \neq \emptyset$ sinon $\beta_j = t_j/(t_j - 1)$	pour $1 \leq j \leq t$

Chaque dent intersecte t_j manches avec $t_j \geq 1$. Chaque manche intersecte un

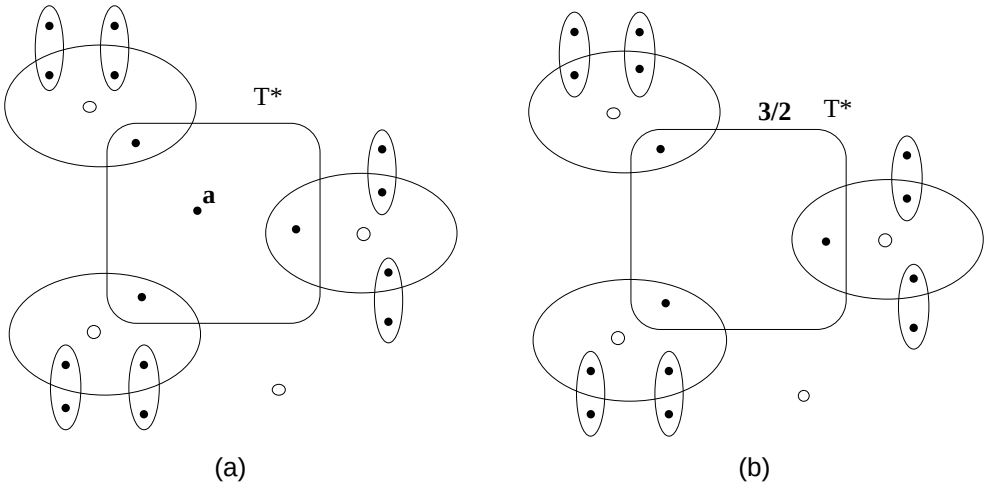

FIG. 2.5 – *un exemple d'arbre de clique et de bipartition* ($h = 3, t = 7$)

nombre impair $h_i \geq 3$, de dents. L'inégalité de bipartition s'écrit comme suit :

$$\sum_{i=1}^{h} x(\delta(H_i)) + \sum_{j=1}^{t} \beta_j x(\delta(T_j)) \geq \sum_{i=1}^{h} (h_i + 1) + 2 \sum_{j=1}^{t} \beta_j$$

Ces inégalités ont été introduites par Boyd and Cunningham [BC91] généralisant les inégalités des *arbres de cliques* (*Clique Tree inequalities*) définies par Grötschel and Pulleyblank [GP86]. Les inégalités d'arbres de cliques ne possèdent pas de dent dégénérée, c'est à dire une dent T_j telle que $T_j \setminus (\bigcup_{i=1}^{h} H_i) = \emptyset$. La figure 2.5 montre un exemple d'arbre de clique et de bipartition.

2.3.6 Autres Inégalités

Comme on ne connaît pas toutes les facettes du $STSP(n)$, on essaye de définir quelques opérations qui permettent la création de nouvelles inégalités à partir des inéquations déjà bien caractérisées. Parmi ces opérations on site *la 2-somme* de 2 inéquations de chemins, *le clonage des arêtes* et *le lifting des sommet* (ou *zero node-lifting*).

Ces opérations ont permis de mettre en évidence quelques relations entre les inéquations écrites sous la forme triangulaire serrée et d'autres inégalités.

Chapitre 3

La méthode du “Branch and Cut”

Dans ce chapitre nous détaillerons toutes les étapes de l’algorithme du “Branch and Cut” pour résoudre le problème du voyageur de commerce que nous avons implémenté. Tout d’abord, nous présenterons un organigramme général de l’algorithme. Ensuite nous décrirons les phases d’initialisation, et de résolution du programme linéaire. Puis nous présenterons la phase de génération de coupes polyédrales avec les heuristiques de séparations et par séparation de pool. Nous décrirons ensuite l’étape de pricing et de génération de colonnes, et nous montrerons les possibilités de fixation de variables pendant cette étape. Nous terminerons enfin, par une présentation des stratégies de branchement et de parcours de l’arbre du “Branch and Cut”.

3.1 Introduction

Notre implémentation est en partie basée sur les travaux de Jean Maurice Clochard et Denis Naddef [CN93], ainsi que sur les travaux de Padberg et Rinaldi [PR91].

La méthode du “Branch and Cut” que nous présentons est une combinaison de la méthode des coupes polyédrales, du “Branch and Bound” et de la méthode des générations de colonnes. Comme toute méthode énumérative implicite, l’algorithme construit une arborescence nommée *l’arbre du “Branch and Cut”*, les sous-problèmes qui forment l’arbre sont appelés *des nœuds*. Il existe trois types de nœuds dans l’arbre du “Branch and Cut”, le nœud courant qui est en train d’être traité, les nœuds actifs qui sont dans la liste d’attente des problèmes et les nœuds inactifs qui ont été élagués au cours du déroulement de l’algorithme.

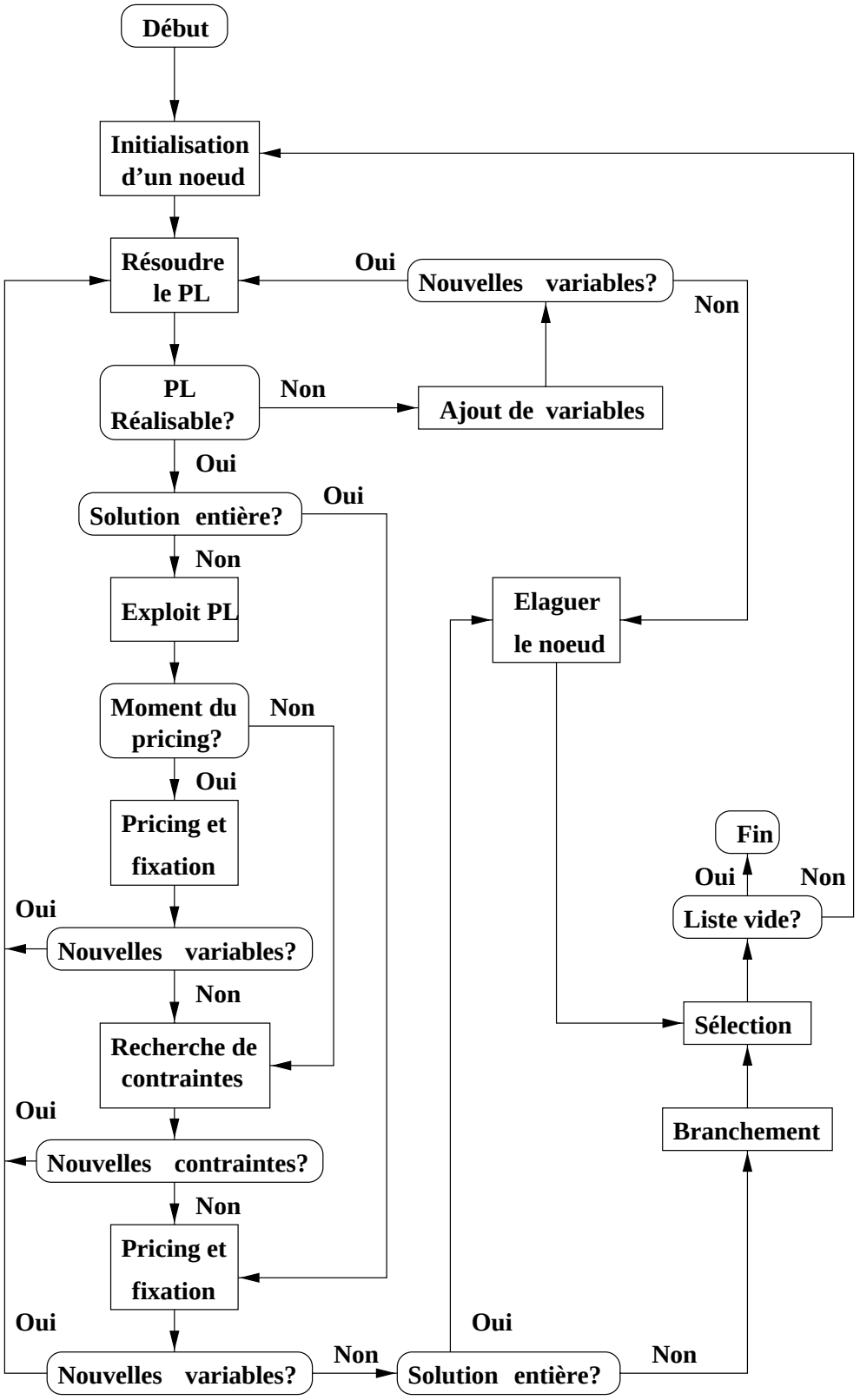

FIG. 3.1 – *l’organigramme du “Branch and Cut”*

Avant d'entrer dans les détails de l'algorithme rappelons et donnons quelques définitions.

Les variables globales $valpl$, bil et bsg désignent respectivement la valeur de la solution optimale du dernier programme linéaire résolu, la borne inférieure du nœud courant et la valeur courante de la meilleure solution réalisable connue (ou borne supérieure globale). Notons que les deux premières variables peuvent être différentes. Comme nous le verrons dans la suite, la borne inférieure d'un nœud est valide pour le graphe complet alors que la valeur de la solution du programme linéaire est valide sur le graphe partiel sur lequel le programme linéaire est résolu. Nous noterons aussi par big la borne inférieure globale, c'est à dire le minimum des bornes inférieures des nœuds actifs et du nœud courant. La variable globale bip dénotera la borne inférieure trouvée lors du calcul du nœud racine de l'arbre du "Branch and Cut" restant. Nous appelons *nœud racine de l'arbre du "Branch and Cut" restant* le premier ancêtre commun dans l'arbre du "Branch and Cut" de tous les nœuds actifs. Comme dans le "Branch and Bound", nous élaguons un sous problème si la borne inférieure locale bil est supérieure ou égale à la borne supérieure globale bsg ou si le sous problème n'est pas réalisable. Nous appelons *variable* du programme linéaire les arêtes du graphe.

L'algorithme du "Branch and Cut" (figure 3.5) est composé de plusieurs phases que nous allons décrire dans la suite : Initialisation, Séparation, Recherche de bornes supérieures, " Pricing " et Fixation, Branchement et Sélection.

3.2 Initialisation

Au début de l'algorithme de résolution du problème du voyageur de commerce, on initialise le programme linéaire par les contraintes de degrés sur tous les sommets du graphe, ainsi que les bornes sur les variables, et on relaxe les contraintes d'intégralités sur les variables et les contraintes de sous-tours, dont le nombre est en $\mathcal{O}(2^n)$.

Le nœud racine de l'arbre du "Branch and Cut" est donc initialisé par le programme linéaire suivant:

$$
\begin{array}{lrl}
\text{minimiser} & c^T x & \\
\text{sous} & x(\delta(v)) = 2 & \text{pour tout } v \in V_n \\
 & 0 \leq x_e \leq 1 & \text{pour tout } e \in E_n
\end{array}
$$

D’autre part, le graphe complet K_n, sur lequel le problème du voyageur de commerce est résolu, possède $\frac{n(n-1)}{2}$ arêtes. Ainsi pour les instances avec 100 villes, le programme linéaire possède 4950 variables et pour des instances avec 1000 villes le programme linéaire possède 499500 variables. Il est donc très difficile, voire impossible, de faire tenir plusieurs programmes linéaires de telle taille dans une mémoire, et aberrant de prétendre pouvoir résoudre des centaines, voire des milliers de fois ces programmes linéaires en un temps raisonnable. D’où l’idée de résoudre le problème du voyageur de commerce sur un petit ensemble de variables choisi d’une façon judicieuse. En effet, sachant que la solution optimale du problème du voyageur de commerce ne contient que n variables, et que souvent les arêtes qui la composent sont incidentes à des sommets qui appartiennent au même voisinage, alors nous pouvons écrire le programme linéaire initial sur un graphe partiel du graphe complet K_n. Nous avons constaté que le graphe des 10 plus proches voisins de K_n a des fortes chances de contenir la solution optimale, alors que celui des 5 plus proches voisins contient une grande partie de cette solution. On peut donc initialiser le programme linéaire soit par le graphe partiel $G = (V, E)$ de K_n des k_s plus proches voisins, soit par le graphe de triangulation de Delaunay.

Pour s’assurer que le graphe de départ contient une solution réalisable on ajoute un cycle hamiltonien trouvé par une heuristique. La borne supérieure globale bsg peut être initialisée par la valeur de ce cycle. Les variables qui forment les colonnes du programme linéaire seront appelées variables actives, et par opposition les autres arêtes de K_n seront appelées des variables inactives. D’autre part, on génère un autre graphe partiel appelé graphe de réserve, des k_r plus proche voisins, $k_r > k_s$. Les variables appartenant à ce graphe de réserve sont inactives, et peuvent être utilisées avec une haute priorité lors des procédures de génération de colonnes. Au début de l’exécution de l’algorithme du “Branch and Cut”, la liste des nœuds actifs est vide.

3.3 Résolution du programme linéaire

La résolution des programmes linéaires est l’un des goulots d’étranglement de la méthode du “Branch and Cut”. On peut passer plus du trois quarts de temps de calcul dans cette procédure pour certaines instances du problème du voyageur de commerce. Il existe maintenant plusieurs logiciels de résolution de programme linéaire, comme Cplex [Cpl95] et OSL [IBM95], qui implémentent la méthode du simplexe (pour plus de détails voir [Chv83]) et la méthode des points inté-

rieurs (pour plus de détails voir [LMS94]). La méthode des points intérieurs est très performante dans la résolution de grandes instances et n'est pas sensible à la dégénérescence des programmes linéaires mais elle ne permet pas l'utilisation de routines efficaces de ré-optimisations. Ces routines sont très rapides dans la méthode du simplexe.

Dans notre implémentation les programmes linéaires sont résolus par la méthode du simplexe implémentée dans Cplex [Cpl95]. En effet, la méthode du "Branch and Cut" nécessite l'utilisation d'un solveur qui possède des routines de ré-optimisation très rapides. Ces méthodes évitent de recommencer toutes les étapes de résolution du programme linéaire dès qu'on ajoute de nouvelles contraintes ou de nouvelles colonnes, ce qui se produit assez souvent dans notre algorithme.

Un programme linéaire est résolu par la méthode primale du simplexe quand on part avec une base réalisable pour le programme primal, et par la méthode duale du simplexe quand on part avec une base réalisable pour le dual. Ainsi, après un ajout de nouvelles contraintes dans le programme linéaire, on utilise la méthode duale pour résoudre le nouveau programme.

3.4 Séparation

Trouver des inégalités violées est une partie cruciale de notre algorithme. La génération de ces inégalités se fait en utilisant les algorithmes de séparation pour chaque type de facettes du $STSP(n)$. Cependant, il n'est pas nécessaire d'appeler à chaque itération les algorithmes de séparation de toutes les classes de facettes. Il est plutôt préférable de les utiliser d'une manière hiérarchique. Ainsi, à cause de leur coût prohibitif en temps de calcul, certains algorithmes de séparations ne sont exécutés que si les autres échouent à trouver des inégalités violées. On développera dans la suite l'idée centrale de certains algorithmes de séparation pour quelques classes d'inégalités du $STSP(n)$.

3.4.1 Séparation par pool

Il est nécessaire d'éliminer de temps en temps des contraintes du programme linéaire, puisque si on ajoute des inéquations sans en éliminer d'autres à la matrice des contraintes du programme linéaire, la taille de cette matrice devient très grande. Et il devient ainsi très difficile de résoudre le programme linéaire en un temps raisonnable, et de trouver de l'espace mémoire pour continuer l'exécution.

On peut stocker un certain nombre des contraintes éliminées dans un pool que l’on peut utiliser pour générer des plans de coupes. En effet, il est possible de générer facilement des contraintes violées par la solution courante du programme linéaire à partir d’un pool de contraintes qui serait construit au fur et à mesure du déroulement de l’algorithme.

Un critère assez simple pour effectuer l’élagage des contraintes du programme linéaire est d’éliminer les inégalités qui ne sont pas actives, c’est à dire dont les variables d’écarts sont positives (les inégalités sont supposées être sous la forme $Ax \leq b$ au départ). Mais pour éviter de cycler, à cause de l’élimination de contraintes qui seront violées tout de suite après dans le programme linéaire suivant, on définit des stratégies d’éliminations plus contraignantes. Une première consiste à éliminer une inégalité dans le cas où sa variable d’écart est supérieure à une certaine valeur. On peut aussi éliminer uniquement les contraintes qui ne sont pas serrées depuis quelques itérations. Une autre stratégie consiste à éliminer les contraintes dont les variables d’écarts sont de base. On peut aussi considérer qu’on ne peut éliminer des contraintes qu’à partir d’une certaine augmentation de la valeur de la solution du programme linéaire en une ou plusieurs itérations.

La séparation par pool implique d’avoir des algorithmes efficaces pour tester rapidement si une contrainte est violée ou non, et surtout pour pouvoir réécrire facilement une contrainte sur le graphe partiel du programme linéaire. Utiliser le pool peut être plus avantageux pour certaines classes de contraintes malgré l’existence d’heuristiques de séparation. Une inégalité peut être violée par la solution du programme linéaire alors que l’heuristique de séparation de ce type de contrainte n’arrive pas à la générer. Cette inégalité pourrait être dans le pool des contraintes si elle était générée dans un autre nœud qui possédait une solution plus adaptée à l’heuristique de séparation. Cette contrainte serait alors générée très facilement du pool et ajoutée au programme linéaire.

Pour ne pas augmenter considérablement la taille du pool, il faut choisir les classes de contraintes qu’on doit stocker et procéder de temps à autres à leurs éliminations du pool. Il est avantageux de sauvegarder d’une manière hiérarchique les contraintes difficiles à générer par les autres méthodes de séparation. Il est évident que les contraintes de sous-tours ne seront jamais sauvegardées dans le pool puisqu’elles sont faciles à générer par un algorithme exact, comme on le verra dans la suite.

3.4.2 Arrêt des séparations

La méthode des coupes polyédrales ne s'arrête pas tant qu'il y a des contraintes violées générées. Or il est plus raisonnable d'arrêter les séparations dans un nœud de l'arbre du "Branch and Cut" s'il n'y a pas eu d'augmentation significative de la valeur du programme linéaire $valpl$ pendant les dernières itérations. On appellera ce phénomène *essoufflement*. Pour cela, on définit une fonction essoufflement qui décide du moment où il faut arrêter de générer des contraintes et passer à la phase de branchement. Le bon réglage de la fonction d'essoufflement est très important. Une fonction très lâche peut faire augmenter considérablement la taille de l'arbre du "Branch and Cut", l'inverse peut augmenter inutilement le temps de résolution d'une instance du problème du voyageur de commerce. En général, on choisit de pousser le plus loin possible la génération de coupes au nœud racine de l'arbre du "Branch and Cut". Ensuite on adopte une fonction d'essoufflement plus lâche.

3.4.3 Séparation des inégalités de sous-tours

Dans notre algorithme il est impératif qu'aucune contrainte de sous-tour ne soit violée avant de passer à d'autres algorithmes de séparation ou à d'autres phases du "Branch and Cut".

La séparation des inégalités du sous-tour est relativement simple. Il suffit de chercher la coupe minimale dans le graphe $G_x = (V, E, x)$ dont l'ensemble des arêtes contient toutes les arêtes de E qui ont une composante non nulle dans x, vecteur d'incidence de la solution courante du programme linéaire. Le poids d'une arête e de G_x est égale à x_e. Soit la coupe $\{S, V \backslash S\}$ tel que $x(\delta(S))$ est minimum. Si la coupe minimale a une valeur inférieure à 2 alors l'inégalité définie par cette coupe est violée par x sinon il n'existe aucune inégalité de sous-tour violée par x.

La séparation des inégalités de sous-tour peut être effectuée avec les algorithmes de recherche de coupe de poids minimum [NI92, PR90]. Quand on ne génère plus d'inégalité du sous-tour violée, on a la garantie que la solution du programme linéaire est soit non entière, soit un vecteur d'incidence d'un cycle Hamiltonien.

Avant de présenter les heuristiques de séparation associées à d'autres classes d'inégalités, nous allons introduire quelques concepts nécessaires à la compréhension de ces procédures.

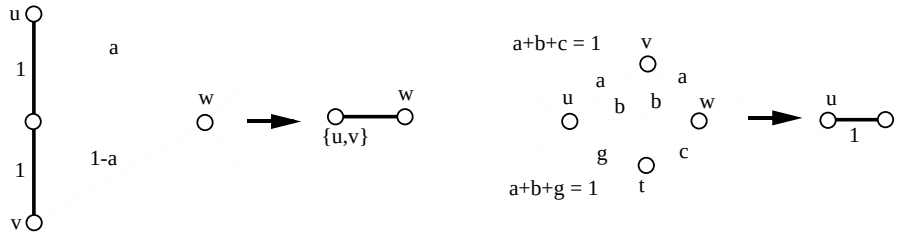

FIG. 3.2 – *Exemples de contractions valides.*

3.4.4 Contractions valides

Le graphe obtenu en contractant (*shrinking*) un ensemble $S \in V$ d'un graphe $G = (V, E)$ est le graphe dont l'ensemble des sommets est $V \backslash S + \{s\}$, où s est un nouveau sommet qui représente les sommets de S, l'ensemble des arêtes est $E' = \{e = (u, v) \in E : u \notin S, v \notin S\} \cup \{(s, v) : v \notin S$ et tel que $\exists t \in S$ tel que $(t, v) \in E\}$ où chaque arête (s, v) a un coût égal à la somme des coûts des arêtes qu'elles représentent.

On dit qu'une contraction est valide (ou légale) pour x, si elle vérifie la propriété suivante : x viole une inégalité valide si et seulement si la solution induite dans le graphe contracté la viole aussi. On peut aussi définir des contractions qui sont valides pour une classe d'inégalité particulière. Ainsi, une contraction est valide pour les inégalités de peigne si elle garantit que x viole un peigne si et seulement si la solution induite sur le graphe contracté le viole aussi. La contraction d'un graphe simplifie considérablement la recherche de contraintes violées. D'où l'importance d'effectuer le maximum de contractions valides avant d'entrer dans les procédures de séparation.

L'une des contractions possibles est celle des chaînes de 1, ou 1-chaîne, c'est à dire les chaînes dont toutes les arêtes e ont un coût $x_e = 1$ dans le graphe G_x. Une 1-chaîne peut être remplacée par une arête en contractant tous les sommets de la chaîne sauf une de ses extrémités.

La deuxième contraction possible porte le nom de *réduction* $(a, 1-a)$ (voir la figure 3.2): soit une chaîne de 1 dont les deux extrémités u et v sont adjacentes au même sommet w tel que $x_{uw} + x_{vw} = 1$. On contracte la chaîne en un sommet s, ce qui induit une nouvelle arête (s, w) tel que $x_{sw} = 1$. Notons que cette contraction peut induire d'autres contractions.

La troisième contraction est moins importante puisqu'elle se produit rarement. Elle concerne un ensemble S de trois sommets qui forment un triangle dans G_x

de coût 1 et qui sont adjacents à un sommet incident à trois arêtes dont la somme des coûts respectifs vaut 1 (voir la figure 3.2).

3.4.5 Construction des cocycles

On a vu que certaines contraintes peuvent être décrites par leur famille $\mathcal{S} = \{S_1, \cdots, S_t\}$ formée de collection de sous-ensembles de V_n. Ces ensembles doivent avoir un cocycle de poids faible pour qu'ils aient une chance d'appartenir à la famille qui forme une inégalité violée. Nous allons donner deux heuristiques de construction de tels ensembles qui possèdent un cocycle de poids faible.

Ordre "Max-Back"

Soit un graphe $G = (V, E)$ avec une fonction poids x sur les arêtes. Pour un sous-ensemble de sommet X et Y soit :

$$c(X, Y) = \sum (x_{uv} : uv \in E, u \in X, v \in Y)$$

Pour un ordre $v_1, \cdots, v_n$ des sommets de G, on note V_i l'ensemble des i premiers sommets. On dit qu'un ordre est " Max-Back " si :

$$c(v_i, V_{i-1}) = \max_{j \geq i} c(v_j, V_{i-1}) \text{ pour tout } i \geq 2$$

Soit S un ensemble de sommets et $v \notin S$, la quantité $c(v, S)$ est appelée la valeur max-back de v ou la valeur avec laquelle v voit S. En d'autres termes dans un ordre max-back chaque sommet voit l'ensemble formé par ses prédécesseurs par une valeur supérieure ou égale aux valeurs avec lesquelles les sommets qui le suivent voient le même ensemble.

Construire un ensemble en utilisant un ordre max-back revient à choisir un sommet de départ puis introduire de nouveaux sommets selon un ordre max-back. Ceci permet de contrôler le poids du cocycle. Dans le cas d'une égalité de la valeur max-back de deux sommets on choisit le sommet qui voit S par le plus grand nombre d'arêtes, si les deux sommets voient S par le même nombre d'arêtes, on choisit le sommet le plus proche de l'ensemble courant en terme de nombre d'arêtes après avoir éliminé les arêtes le connectant directement à S.

Remarquons que l'ordre max-back est l'idée centrale de l'algorithme de coupe minimum de Nagamochi et Ibaraki [NI92]. En effet, si on note $\lambda(u, v)$ la valeur

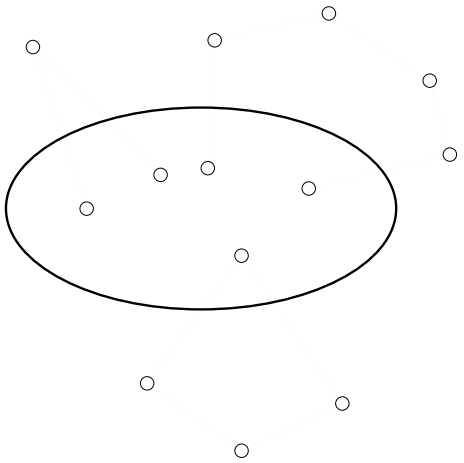

FIG. 3.3 – *Exemple d’oreille.*

de la coupe minimum séparant u et v dans un graphe G, on a le théorème suivant :

Théorème 6

$$\lambda(v_n, v_{n-1}) = c(v_n, V_{n-1}) \ (= c(v_n))$$

L’algorithme de Nagamochi et Ibaraki est basé sur l’observation suivante : soit G' le graphe obtenu à partir de G en contractant u et v. La coupe minimum dans G, $\lambda(G)$, vérifie:

$$\lambda(G) = min(\lambda(u, v), \lambda(G'))$$

Pour deux sommets u et v tel que $d(u) = \lambda(u, v)$ on a $\lambda(G) = min(\lambda(G'), d(u))$, d’où pour déterminer $\lambda(G)$ il suffit de calculer la valeur de la coupe $\lambda(G')$ du graphe contracté G'. Le théorème nous indique comment trouver des paires de sommets u,v.

Construction par oreille

Soit le sous graphe de $G = (V, E)$ induit par un sous-ensemble de sommets S. Une oreille est un chemin extérieur minimal entre deux sommets $u \notin S$ et $v \notin S$, incidents à S c’est à dire un chemin sans corde et tel que les sommets appartenant à ce chemin sont tous dans $V \backslash S$ (voir figure 3.3). On agrandit un ensemble de sommets en choisissant un sommet de départ ensuite en introduisant

à chaque itération les sommets d'une oreille qui minimise la valeur du cocycle. Cette méthode permet de mieux contrôler la façon de construire l'ensemble des sommets.

3.4.6 Séparation des peignes

Pour trouver des inégalités de peignes violées, il suffit de trouver un ensemble $\mathcal{S} = \{H, T_1, \cdots, T_{2k+1}\}$ c'est à dire un manche H et un nombre impair de dents qui dépend de la valeur du cocycle $x(\delta(H))$, et qui vérifient les conditions suivantes:

$$\begin{array}{lll} (i) & \forall i = 1, ..., 2k+1 & H \cap T_i \neq \emptyset \\ (ii) & \forall i = 1, ..., 2k+1 & T_i \setminus H \neq \emptyset \\ (iii) & \forall 1 \leq i < j \leq 2k+1 & T_i \cap T_j = \emptyset \end{array}$$

Il est facile de constater qu'il faut trouver $2k+1$ dents si le poids du cocycle de H est entre $2k$ et $2k+2$. Notons qu'il existe de faibles chances de trouver des inégalités violées si la valeur du cocycle est proche de ces 2 bornes.

Une première méthode [PR90] consiste à éliminer du graphe G_x toutes les arêtes de valeur 1. Les composantes biconnexes ou leurs unions, sont des manches potentiels. Les dents sont soit les arêtes de poids 1 incidentes au manche, soit d'autres composantes biconnexes. Une autre méthode pour trouver un manche, consiste à construire un ensemble de sommets à partir d'un sommet donné en ajoutant des sommets selon des critères induits par l'ordre max-back ou par addition d'oreilles.

Pour trouver des inégalités de peignes violées, il faut savoir à partir de quel sommet il faut commencer à construire un manche, à quel moment il faut arrêter d'agrandir le manche et commencer à chercher des dents, et quels sont les sommets qui sont des bons candidats pour commencer à chercher des dents.

Recherche d'un manche

Pour construire un manche à partir d'un sommet, il est judicieux de commencer par un sommet qui aurait de fortes chances d'appartenir à une dent d'un peigne violé. Les meilleurs candidats sont les dents dont le cocycle a une valeur proche de 2, rappellons que cette valeur est toujours supérieure ou égale à 2 puisque toutes les contraintes de sous-tour sont satisfaites. Il suffit donc de trouver toutes les

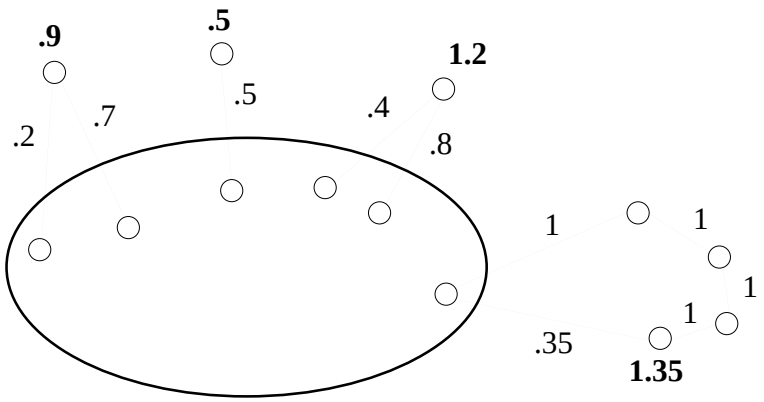

FIG. 3.4 – *Exemples de chaque liste.*

coupes de valeur 2 par un algorithme polynomial [dV97, Fle97] et ensuite choisir un sommet qui appartient à l’un des ces coupes, dont la valeur de max-back est inférieure à une valeur donnée. Ce sommet appartiendra à l’intersection de la dent et du manche, et sera le sommet de départ pour la construction du manche. Ainsi on a déjà une dent dont la valeur du cocycle est la plus petite possible. Dans l’implémentation qu’on utilise, on ne considère que les ensembles formés de 1-chaîne. On choisit de commencer à partir de l’une des 2 extrémités de la chaîne, d’ailleurs on essaye les deux. On interdira toujours à l’autre extrémité d’entrer dans le manche.

Etant donnée le manche courant H, on maintient trois types de listes :

– La liste des sommets qui voient H avec une valeur inférieure à 1.

– La liste des sommets qui voient H avec une valeur supérieure ou égale à 1, sans qu’il y ait d’arête ou chaîne de poids 1 qui relie le sommet au manche.

– La liste des sommets qui voient H avec une valeur supérieure ou égale à 1, mais qui sont incidents à une arête ou à une chaîne de poids 1 qui les relient au manche.

La figure 3.4 donne des exemples de sommets de chaque liste. Les deux premiers sommets appartiennent à la première liste. Le sommet suivant appartient à la deuxième liste et le dernier sommet appartient à la troisième liste.

Les sommets de la deuxième liste qui ont une valeur de max-back strictement supérieure à 1, peuvent être ajoutés automatiquement à H puisqu’ils diminuent le poids du cocycle. Les sommets qui ont une valeur de max-back égale à 1, laissent

la valeur du cocycle inchangée. Ces sommets peuvent être aussi ajoutés automatiquement à H. Quant aux deux autres listes, il est beaucoup plus difficile de décider s'il faut faire entrer les sommets de la troisième liste sous peine de perdre un très bon candidat pour être une dent du peigne ou les sommets de première liste sous peine d'augmenter la valeur du cocycle. Malheureusement il n'existe pas de réponse exacte à ce problème. On peut choisir de faire entrer les sommets de la troisième liste si leur valeur max-back excède une valeur donnée. Alors que pour les sommets de la première liste, leur inclusion dans la manche dépend fortement de leur valeur max-back et leur environnement.

A chaque fois que la deuxième liste est vide et la valeur du cocycle est proche d'un entier impair. On essaye de trouver un peigne violé avec le manche obtenu. Dans le cas d'un échec on continue d'agrandir le manche, jusqu'au moment où celui-ci devient excessivement grand, alors on décide de l'abandonner.

Recherche des dents

Quand on a un manche candidat à appartenir à un peigne violé. On doit commencer à chercher des dents. Leur nombre t est connu, il est égal à $2k + 1$ si la valeur de $x(\delta(H))$ est entre $2k$ et $2k + 2$. S'il existe t arêtes de valeur 1 qui appartiennent au cocycle de H alors il suffit de prendre chacune de ces arêtes comme une dent du peigne et on a une contrainte violée. Rappelons que ces contraintes portent le nom de 2-couplage. Sinon il faut chercher à compléter l'ensemble des dents. Pour ce faire, on choisit un sommet $u \in H$ dont la valeur max-back est minimum, qui n'est pas incident à une arête de valeur 1. Ce sommet pourrait être un bon candidat pour appartenir à l'intersection d'une dent et du manche. Ensuite on cherche un nœud $v \notin H$ adjacent à u tel que x_{uv} est assez grand. La solution la plus simple serait d'appliquer un algorithme de coupe minimum qui sépare $\{u, v\}$ et l'ensemble des sommets qui ne doivent pas appartenir à une dent. Cette méthode bien qu'exacte, est très coûteuse en terme de temps de calcul. Elle pourrait être utilisée quand il devient assez difficile de trouver des contraintes violées. Une méthode heuristique beaucoup plus rapide consiste à construire la dent en partant du sommet u en ajoutant des sommets selon la méthode max-back ou en ajoutant des oreilles comme cela a été décrit pour les manches. Notons qu'on laisse la dent croître librement à l'intérieur du manche. On s'arrête dès qu'on trouve une dent de cocycle raisonnable. On interdit les sommets qui appartiennent à l'intersection de H et de la dent. Et on recommence. Quand on arrive à obtenir le nombre de dents nécessaires on vérifie si la contrainte est violée.

Les choix que nous avons présentés sont des heuristiques, elles sont donc ame-

nées à être améliorées. Ces choix se basent sur beaucoup de bon sens, et nous ont permis de trouver de très bons résultats dans la séparation des peignes. Pour mieux approfondir ces choix vous pouvez consulter [NT98a]. Notons que parmi toutes les classes d'inégalités connues, à part les inégalités de sous-tours, les inégalités de peignes sont probablement les meilleures coupes du polytope du voyageur de commerce. D'où l'importance de la mise en œuvre des heuristiques performantes de séparation pour cette classe d'inégalités.

3.4.7 Autres séparations

Pour effectuer les séparations d'autres types de contraintes, on a utilisé comme point de départ l'algorithme de séparation des peignes. On verra dans la suite comment on peut partir d'un peigne non violé, obtenu par les heuristique de séparation des peignes décrites ci-dessus, pour trouver des contraintes d'arbres de cliques, d'échelles ou de chemins violées.

Séparation des arbres de cliques

On va montrer comment on peut partir d'une inégalité de peigne non violée pour construire une inégalité d'arbre de cliques violée. Si on considère une inégalité de peigne non violée à cause de la grande valeur de l'un des cocycles de ses dents. La figure 3.5 montre comment on peut construire une inégalité d'arbre de clique violée en ajoutant un nombre adéquat de manche qui intersecte la dite dent. En gras sur la figure on voit le peigne non violé de départ.

Ainsi on déduit le principe de l'heuristique de séparation des arbres de cliques. Supposons qu'on possède un ensemble $\mathcal{S} = \{H_1, T_1, \cdots, T_{2k+1}\}$ définissant une inégalité de peigne non violée. En commençant par la dent qui possède le plus grand cocycle, on essaye de construire avec la méthode de l'oreille ou max-back un ou plusieurs manches, en partant d'un sommet de la dent qui possède une valeur max-back minimum par rapport à la dent. Pour chaque manche construit, on cherche des dents pour compléter l'arbre de clique. On répète ceci pour chaque dent du peigne initial dont le cocyle est supérieur à une certaine valeur.

Séparation des inégalités d'échelles

De la même façon que pour les arbre de cliques, on va essayer de trouver des inégalités d'échelles violées à partir de contraintes de peignes non violées. La figure 3.6 montre des exemples de contraintes d'échelle violées obtenues à

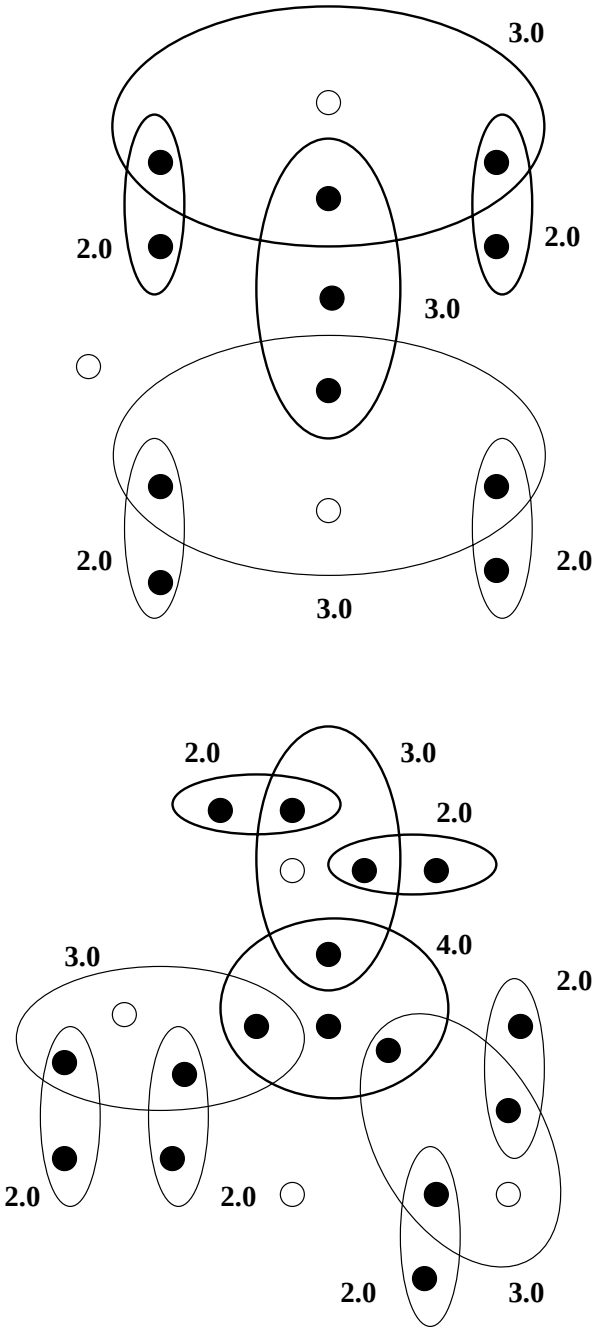

FIG. 3.5 – *Exemple d'inégalités violées d'arbre de clique*

partir de contrainte de peigne. En gras sur la figure, on distingue l'inégalité de peigne d'origine. Les coefficients de chaque dent dans l'inégalité d'échelle est en police fine. Dans les deux premiers cas, le terme correcteur est nul puisque $(H_1 \cap P_1 : H_2 \cap P_2) = \emptyset$.

On va distinguer la méthode de séparation des inégalités d'échelle selon l'origine de la violation. Dans le cas où le terme correcteur n'est pas l'origine de la violation, on commence comme d'habitude par un peigne non violé. Le manche du peigne va être le manche H_1 de l'inégalité d'échelle potentiel. Le point crucial de notre heuristique consiste à choisir la dent pendante P_1 qui n'intersecte pas le manche H_2. Cette dent doit avoir un cocycle proche de 2. Si plusieurs cocycles satisfont cette condition, on choisit la dent qui est la moins reliée aux dents dont la valeur du cocycle est grande. En effet, soient T et T' tels qu'un sommet de $T \setminus H_1$ est adjacent à un sommet de $T' \setminus H_1$ par une arête de valeur non négligeable, et telle que la valeur du cocycle de T' est assez loin de 2, alors ce sommet est un bon candidat pour commencer la construction d'un manche H_2, d'où la dent T n'est pas un bon candidat pour le rôle de P_1.

Une fois décidé quelle dent va jouer le rôle de P_1, il faut construire le manche H_2. Ceci est fait comme d'habitude par construction à partir d'un sommet initial par l'addition d'oreilles ou par la méthode de max-back. Ensuite, il suffit de construire la dent pendante P_2 comme dans les heuristiques de séparation des peignes.

On a vu dans la description des inégalités d'échelle qu'on peut obtenir le même type de contrainte en remplaçant le manche H_2 par son complémentaire. Pour obtenir ce type de contrainte, on procède de la même façon que pour les inégalités classiques d'échelles. La dent du manche intérieur est choisie comme pour la dent pendante P_1. Le manche extérieur est initialisé par l'union du manche intérieure et la dent P_1. Il est construit jusqu'à ce que son cocycle atteint une valeur raisonnable et tel que toutes les dents avec une grande valeur de cocycle aient un sommet dans ce manche qui ne soit pas dans le manche intérieur, pour qu'ils aient un coefficient égale à 1. A ce stade, on cherche une dent P_2 comme pour les peignes.

Séparation des inégalités d'étoile et de chemin

Comme d'habitude on va partir d'inégalité de peigne non violées pour construire des inégalités de chemin ou d'étoile violées. La figure 3.7 montre des exemples de contraintes de chemin violées obtenues à partir de contraintes de peignes. Les coefficients de chaque dent dans l'inégalité de chemin sont en police fine.

Pour construire des inégalité violées, il est donc nécessaire de trouver d'autres

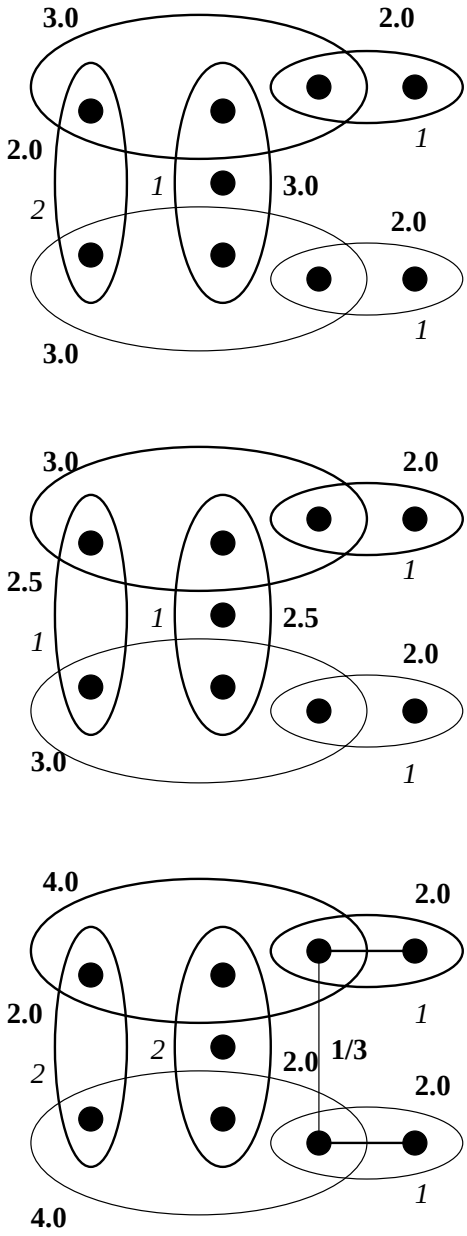

FIG. 3.6 – *Exemple d'inégalités violées d'échelle*

manches tels que leurs sommets soient dans les dents. Une façon de trouver de nouveaux manches est d’agrandir le manche initial en additionnant des oreilles ou des sommets par la méthode de max-back faisant en sorte de rendre les coefficients des dents assez petits. Une autre façon consiste à prendre chaque dent et à essayer de trouver une décomposition en intervalles.

Pour un aperçu plus détaillé des choix d’implémentation des ces procédures de séparation, le lecteur est encouragé de consulter [NT98b].

3.5 Recherche d’une borne supérieure

La recherche de borne supérieure est un point fondamental dans n’importe quel algorithme d’énumération implicite. Une bonne borne supérieure permet d’élaguer rapidement plusieurs branches de l’arbre de recherche.

A chaque itération on vérifie si la solution courante du programme linéaire est un vecteur d’incidence d’un cycle hamiltonien. Dans ce cas si la valeur de la solution est inférieure à la borne supérieure globale bsg alors on met à jour cette variable et on sauvegarde la solution comme étant la meilleure solution obtenue. Sinon, on essaye d’exploiter la solution du programme linéaire afin de construire un cycle hamiltonien de bonne qualité.

La solution du programme linéaire donne parfois une bonne information sur la probabilité d’appartenance d’une arête à la solution optimale. Dans notre heuristique de recherche de solution réalisable, on va considérer qu’une variable e de poids x_e égale ou proche de 1, dans une solution du programme linéaire, a une forte probabilité d’appartenir à la solution optimale. On exploite ainsi ces informations pour construire selon des règles précises un cycle hamiltonien (voir [Bou95]). On applique ensuite un algorithme de Lin-Kerningham modifié pour améliorer la qualité du cycle obtenu. Cette heuristique nous a permis de trouver rapidement des solutions réalisables de très bonne qualité.

3.6 Pricing et Fixation

Puisqu’on travaille sur un sous ensemble d’arêtes tout au long de l’algorithme du “Branch and Cut”, il faut vérifier que la solution du programme linéaire sur le graphe partiel reste optimale pour le graphe complet, avant de pouvoir élaguer ou brancher un nœud de l’arbre du “Branch and Cut”. D’où la nécessité de faire appel à la méthode de génération de colonnes pour effectuer un pricing des va-

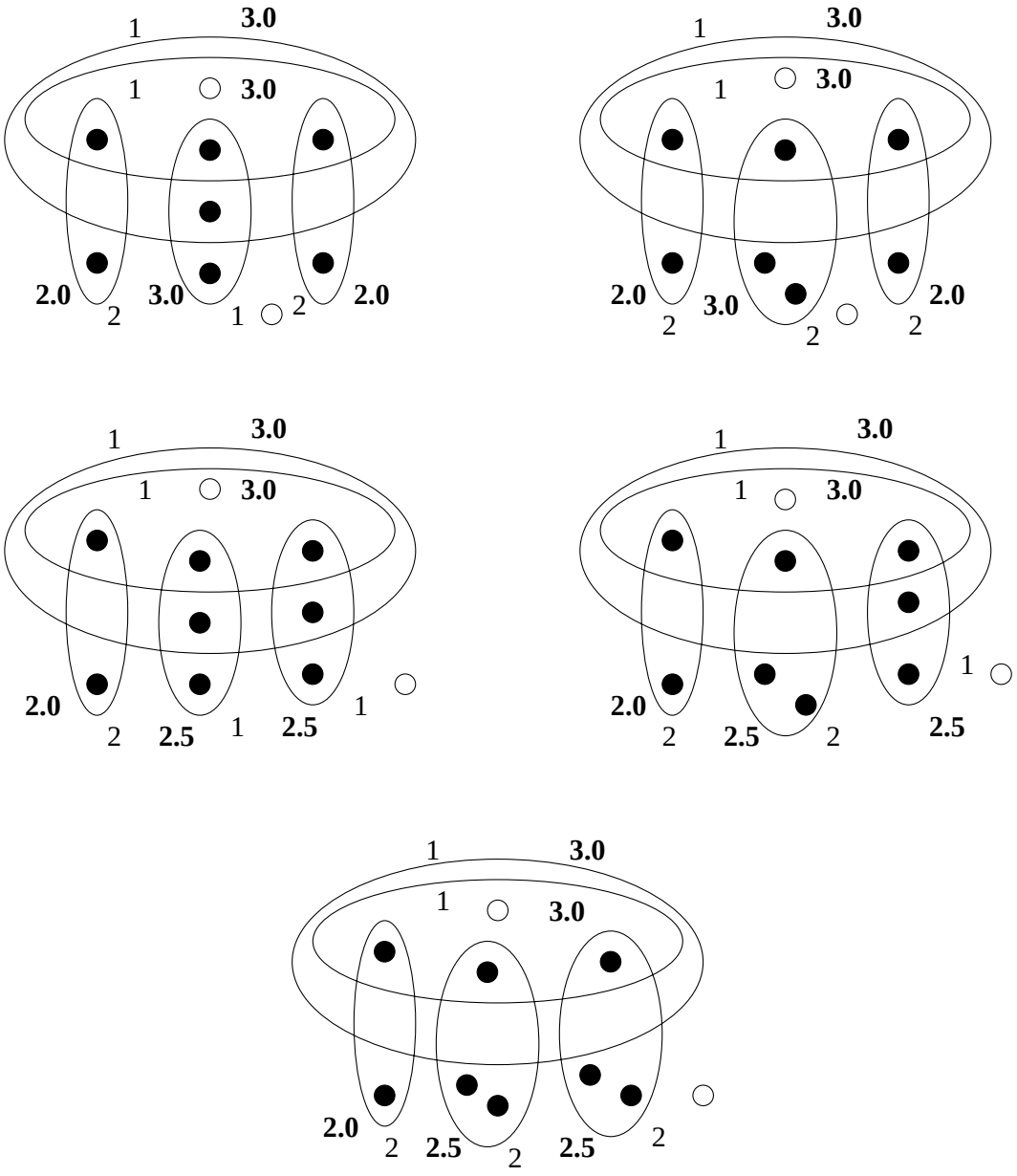

FIG. 3.7 – *Exemple d'inégalités violées de chemin avec deux manches*

riables inactives. Rappelons que la méthode des générations des colonnes, vérifie si toutes les variables inactives ont un coût réduit positif. Dans le cas contraire, elle ajoute au graphe partiel les variables dont le coût réduit r_e est négatif. Le nouveau programme linéaire est résolu avec la méthode primale du simplexe en utilisant la dernière base, puisqu'elle est primale réalisable. A la fin du pricing, on met à jour la valeur de la borne inférieure locale bil et éventuellement la valeur de la borne inférieure globale big.

Pour améliorer les performances du “Branch and Cut”, on effectue un pricing des variables périodiquement, par exemple tout les k programmes linéaires résolus. Ceci permet de faire entrer très tôt des variables inactives dans le graphe partiel nécessaires à la construction de la solution optimale ou à des solutions de bonne qualité.

Au lieu d'appliquer la méthode de génération de colonnes sur toutes les variables inactives, on effectue cette opération d'une façon hiérarchique. Au début, on ne considère que les variables dans le graphe de réserve. Si aucune variable n'est ajoutée au graphe partiel alors on passe au pricing du reste des variables inactives. Cette énumération permet de construire une liste exhaustive des arêtes candidates. Au début de l'exécution de l'algorithme du “Branch and Cut” cette liste est très grande et ne peut pas être sauvegardée. D'où on détermine une taille de buffer raisonnable où une énumération systématique doit être effectuée.

Au fur et à mesure de l'avancement du calcul on peut aussi réduire le nombre de variables inactives à considérer. En effet, si le nœud courant de l'arbre du “Branch and Cut” est une racine de l'arbre restant, et si le coût réduit d'une variable inactive e vérifie la relation $valpl + r_e > bsg$ alors la variable n'a plus à être considérée puisqu'elle est fixée à zéro. Ainsi il y a des fortes chances que la liste complète des variables à considérer pendant le pricing tient dans le buffer très tôt pendant l'exécution de l'algorithme. Ceci permet d'accélérer considérablement l'étape de génération de colonnes.

Pour pouvoir calculer rapidement le coût réduit d'une variable inactive, il faut être capable de calculer le coefficient de chaque variable dans les contraintes actives. Pour ce faire, pour chaque sommet v du graphe, on sauvegarde la liste des contraintes qui contiennent v. Ensuite on calcule le coût réduit d'une variable $e = uv$ en retranchant de c_e le coefficient de l'arête dans une contrainte active, multiplié par la valeur de sa composante dans la solution duale.

3.6.1 Fixation

Avant de faire appel à l'opération de branchement, on essaye de fixer les variables actives hors base en utilisant la valeur de leur coût réduit. En effet étant donné le coût réduit r_e d'une variable e, et sa valeur courante x_e. Si la valeur x_e passe de x_e à $x_e + t$ alors la valeur de la fonction économique augmente de $t \times r_e$. Ainsi on peut fixer

- x_e à 0 si $x_e = 0$ et $bip + r_e > bsg$.
- x_e à 1 si $x_e = 1$ et $bip - r_e > bsg$.

En effet, le coût réduit représente une borne inférieure du changement absolu de la valeur de la fonction économique qui résulte de forcer une variable à passer d'une borne à une autre. D'où, si l'augmentation de la valeur de la fonction économique est assez grande pour causer l'élagage du noeud courant, alors on sait qu'on peut fixer la variable à sa valeur courante.

Les fixations sont valides jusqu'à la fin de l'algorithme si le noeud courant est aussi le noeud racine de l'arbre du "Branch and Cut". Dans le cas contraire les fixations ne sont valides que pour le noeud courant et sa sous arborescence; puisque au contraire de la génération de nouvelles coupes qui ne fait que diminuer la différence entre la borne inférieure et la borne supérieure, la génération de colonnes peut augmenter la valeur de cette différence.

Après les fixations par coût réduit, on peut fixer d'autres variables par implication logique, ceci peut se produire dans les cas suivants:

- Si deux arêtes incidentes à un sommet v ont été fixées à 1 alors toutes les autres arêtes incidentes à v peuvent être fixées à 0. L'arête incidente aux deux autres extremités des deux arêtes peut être fixée à 0.
- Si $n - 3$ arêtes incidentes à un sommet v ont été fixées à 0 alors les deux autres arêtes incidentes à v peuvent être fixées à 1.

3.6.2 Réalisabilité du programme linéaire

Pendant l'exécution du "Branch and Cut", le programme linéaire courant peut être irréalisable pour trois raisons :

Une contrainte de degrés peut être violée quand toutes les arêtes du graphe partiel incidentes à v sauf une, sont fixées à 0. Dans ce cas il suffit d'ajouter de nouvelles arêtes incidentes à v. Si toutes les arêtes incidentes à v sont déjà

présentes dans le graphe partiel alors il faut élaguer le nœud de l'arbre du “Branch and Cut”, puisqu'il n'est pas réalisable.

Une inégalité dans le programme linéaire a sa partie gauche vide, c'est à dire que tout ses variables sont fixées, et elle est violée. Puisque toutes les variables ont des coefficients positifs alors même en ajoutant des variables l'inégalité restera violée. Le nœud de l'arbre du “Branch and Cut” doit être alors élagué.

Dans le cas où aucune des deux premières raisons n'est la cause de l'irréalisabilité du programme linéaire, on effectue une opération de ” pricing ” pour vérifier si la solution réalisable duale est duale réalisable pour tout le problème. On ajoute les variables inactives qui possèdent un coût réduit négatif dans le graphe partiel utilisé pour résoudre le programme linéaire.

Si des variables sont ajoutées, alors on résout de nouveau le programme linéaire. S'il est toujours irréalisable alors on cherchera à le rendre réalisable en ajoutant de nouvelles colonnes avec des critères plus sophistiqués que le simple signe du coût réduit des variables. Notons au passage que la valeur du programme linéaire $valpl$ est une borne inférieure de la valeur de la fonction économique du nœud de l'arbre du “Branch and Cut”. Ainsi, si $valpl \geq bsg$ alors on peut élaguer le nœud.

On essaye d'ajouter des variables parmi celles qui ne sont pas réalisables pour rendre le programme linéaire réalisable. Une variable e dont la valeur dans le programme linéaire est x_e est dite irréalisable si elle viole sa borne inférieure ou sa borne supérieure. C'est à dire $x_e < 0$ ou $x_e > 1$. Une variable d'écart est dite irréalisable si elle a une valeur négative dans le programme linéaire. Une variable e de coût réduit r_e est candidate seulement si $valpl + r_e \leq bsg$. Soit B la matrice de base qui correspond à la solution duale réalisable et primale non réalisable. Pour chaque variable candidate, soit a_e la colonne de la matrice de contraintes correspondant à e et $\overline{a}_e$ solution du système $B\overline{a}_e = a_e$. Soit $\overline{a}_e(b)$ la composante de $\overline{a}_e$ correspondant à la variable de base irréalisable x_b. L'augmentation de x_e peut rendre le programme linéaire réalisable si et seulement si

– x_b est une variable structurelle c'est à dire correspondant à une arête de G et

$$x_b < 0 \text{ et } \overline{a}_e(b) < 0$$

ou

$$x_b > 1 \text{ et } \overline{a}_e(b) > 0$$

– x_b est une variable d'écart et

$$x_b < 0 \text{ et } \overline{a}_e(b) < 0$$

3.7 Branchement

Pendant la procédure de branchement, un nœud de l'arbre du "Branch and Cut" est subdivisé en deux ou plusieurs sous-problèmes, de telle façon que l'union de l'ensemble des solutions de ces sous-problèmes soit égale à celle du nœud père. Il existe plusieurs stratégies de branchements, mais la plus répandue est celle qui consiste à choisir une variable dont la valeur est fractionnaire et à la forcer à être fixée à 0 dans un nouveau sous-problème et à 1 dans l'autre. Pour le problème du voyageur de commerce, si la solution n'est pas réalisable alors il existe forcement une variable fractionnaire, car on utilise une séparation exacte des contraintes du sous-tours.

Il existe plusieurs stratégies pour choisir une variable de branchement. Nous présenterons ici quelques-unes de ces stratégies:

– Choisir la variable dont la valeur est la plus proche de 0.5.

– Choisir la variable fractionnaire qui a le plus grand coefficient dans la fonction économique.

– Choisir parmi les variables fractionnaires dont la valeur est proche de 0.5 celle qui possède le plus grand coefficient dans la fonction économique. Soit $L = min\{0.5, max\{\overline{x_i} : \overline{x_i} < 0.5, i \in I\}\}$ et $H = max\{0.5, min\{\overline{x_i} : \overline{x_i} > 0.5, i \in I\}\}$. L'ensemble des variables candidates pour le branchement est égal à $\{i \in I : 0.75 * L < \overline{x_i} < 0.25 + 0.75 * H\}$. Parmi ces variables ont choisit celle qui possède le plus grand coefficient dans la fonction économique.

– S'il existe des variables fractionnaires qui appartiennent à la meilleure solution réalisable connue alors choisir celle qui a le plus grand coefficient dans la fonction économique. Sinon utiliser la stratégie précédente.

– Cette stratégie est utilisé dans "Cplex callable library" [Cpl95], et porte le nom de *"branchement fort"* (*strong branching*). Sélectionner un ensemble de variables candidates au branchement. Et résoudre pour chaque variable deux programmes linéaires. Un avec la variable fixée à zéro et l'autre avec

la variable fixée à un. Les programmes linéaires ne sont pas résolus nécessairement jusqu'à l'optimalité, en effet on peut diminuer le temps de calcul en imposant un nombre d'itérations limité à effectuer par le simplexe. La variable de branchement est choisie ensuite, en comparant les valeurs des solutions des programmes linéaires.

Les stratégies de choix diffèrent selon le but recherché. On peut soit chercher à élaguer rapidement des sous-problèmes en choisissant la variable de branchement. Soit essayer de choisir la variable qui a la plus forte chance d'être (ou de ne pas être) dans la solution optimale.

Au lieu de partitionner un problème en branchant sur une variable il est aussi possible de brancher sur une contrainte. Naddef et Clochard [CN93] ont proposé de brancher sur les contraintes de sous-tours. Sachant que pour toute solution réalisable du problème du voyageur de commerce on a:

$$x(\delta(S)) \text{ est pair, pour tout } S \subset V_n, S \neq V_n, S \neq \emptyset$$

Alors, si on trouve un ensemble de sommet S telle que le cocycle $x(\delta(S))$ est proche d'un nombre impair $2k+1$, avec $k \geq 1$, le branchement peut être effectué en imposant $x(\delta(S)) \leq 2k$ sur un côté et $x(\delta(S)) \geq 2k+2$ de l'autre. Cette méthode de branchement a permis de diminuer considérablement le temps de résolution de plusieurs instances du problème du voyageur de commerce. La recherche d'un ensemble S candidat au branchement peut être faite durant les séparations, en prenant les manches des inégalités de peigne ou de chemin. Dans notre implémentation, le choix de l'ensemble S sur lequel on doit brancher la contrainte du sous-tour s'effectue selon la même stratégie de “branchement fort” décrite pour les variables. Notons que le branchement sur les contraintes de sous-tours est très performant quand la valeur de la borne inférieure est relativement mauvaise (très loin de la valeur de la solution optimale!), mais leur performance se dégrade quand la borne inférieure devient très proche de la valeur de la solution optimale car il devient très difficile de trouver des cocycles de valeurs impaires. A ce moment, on se tourne vers des stratégies de branchement sur des variables.

Notons aussi qu'il est possible d'élaguer un nœud au moment de la résolution du programme linéaire pendant la phase de branchement. Dans le cas où la fixation d'une variable à l'une de ses bornes induit l'élagage du noeud, il suffit de fixer cette variable à sa borne complémentaire et d'élaguer tous les nœuds où elle est fixée à la même valeur.

3.8 Stratégie de parcours

Le choix d'un nœud actif dépend de la stratégie de parcours adoptée. La stratégie de sélection est un point crucial au succés de l'algorithme. Les quatre stratégies les plus connues sont les suivantes:

- Meilleur d'abord: on choisit le nœud de meilleure évaluation dans la liste des nœuds actifs. Pour le problème du voyageur de commerce on choisit donc le nœud dont l'evaluation est minimum.
- Profondeur d'abord: on choisit un fils du dernier nœud évalué. Si aucun de ces nœuds n'est actif, on "backtrack" dans l'arbre.
- Largeur d'abord: On traite les nœuds dans l'ordre de leur création.
- Mixte: on commence par un parcours en profondeur d'abord sur la premiere branche de l'arbre puis on continue par la stratégie meilleur d'abord.

Le choix d'une stratégie dépend des contraintes sur l'espace mémoire, le temps d'exécution et la connaissance d'une solution réalisable du problème. Si la stratégie en profondeur nécessite moins d'espace mémoire puisque le nombre des nœuds actifs reste faible, la stratégie meilleur d'abord permet de diminuer le temps de calcul puiqu'elle ne visite qu'un nombre minimum de noeud de l'arbre complet d'énumération. Notons qu'on utilise la stratégie profondeur d'abord surtout pour avoir rapidement une solutions réalisables du problème et donc une borne supérieure. Ceci n'est pas nécessaire dans notre implémentation puisqu'on sait trouver rapidement des solution réalisable de très bonne qualité grâce à notre heuristique d'exploitation des solutions fractionnaires du programme linéaire. Notons pour terminer que la stratégie largeur d'abord est tombée en désuétude, en raison de sa faible efficacité.

Chapitre 4

Quelques notions de parallélisme

Dans ce chapitre nous introduirons quelques concepts de base du parallélisme, nous présenterons la problématique de conception d'algorithme parallèle pour des applications irrégulières. Nous terminerons par les fonctions d'évaluation d'un algorithme parallèle.

4.1 Introduction

Les applications informatiques nécessitent de plus en plus de puissance de calcul. Pour obtenir des niveaux de performance élevés, on peut construire des systèmes informatiques dans lesquels plusieurs unités de calcul coopèrent pour résoudre le même problème. Un travail avance plus vite à plusieurs que seul et il est possible de réaliser à plusieurs des travaux dont la taille dépasse les capacités d'un seul.

L'évolution des machines parallèles a suivi l'évolution des processeurs et des réseaux. Des processeurs de plus en plus puissant sont apparus et ont permis de construire des stations de travail avec des puissances équivalentes à celles des premiers supercalculateurs. Les premiers supercalculateurs ont été construits autour des microprocesseurs vectoriels. Cette famille de microprocesseurs "pipeline" les opérations élémentaires pour calculer rapidement des opérations d'algèbre linéaire courantes. Les machines Cray ont largement exploité cette technique. Mais le prix des machines à base de microprocesseurs vectoriels reste trop élevé.

L'apparition des réseaux à haut débit a permis d'interconnecter des dizaines, voire des centaines d'ordinateurs qui échangent des informations à vitesse très élevée, ce qui a permis de voir une nouvelle génération de machines parallèles

à faible coût qui utilisent des composants standards tant au niveau des systèmes d'exploitation que du réseau d'interconnexion et des processeurs.

4.2 Architecture des machines parallèles

Les ordinateurs parallèles sont construits autour d'un ensemble de plusieurs nœuds interconnectés. Chaque nœud correspond à un processeur de calcul qui peut communiquer avec les autres processeurs. Le processeur est relié à un espace mémoire qui peut être partagé avec les autres nœuds.

Les machines parallèles ont été classées selon leur mode de fonctionnement. Flynn [Fly79] distingue quatre types de machines selon le nombre de flot d'instruction et de données :

- Les machines SISD (*Single Instruction Single Data*): Une seule instruction s'exécute sur une seule donnée à la fois. Ce type d'architecture correspond à une machine monoprocesseur.

- Les machines SIMD (*Single Instruction Multiple Data*): Les processeurs exécutent la même instruction sur des données différentes. Ce type de machine est utilisé pour des calculs spécialisés. Elles peuvent contenir plusieurs centaines de processeurs grâce à la simplicité de leur architecture.

- Les machines MISD (*Multiple Instruction Single Data*): plusieurs instructions s'exécutent sur une seule donnée. Ce type de machine n'a pas d'éxistance réelle.

- Les machines MIMD (*Multiple Instruction Single Data*): Ce sont les machines les plus courantes aujourd'hui. Ce type d'architecture représente un système de processeurs autonomes où chacun exécute son propre flot d'instructions sur des flots de données différentes.

Ducan [Duc90] a étendu cette classification selon le mode de synchronisation des nœuds :

- Les machines synchrones : Une horloge globale synchronise les différents nœuds du système sur une base de temps commune.

- Les machines asynchrones : Chaque machine est autonome et possède une horloge locale qui lui est propre.

Les machines SIMD sont synchrones par définition. La majorité des machines MIMD sont asynchrone. On s'intéresse ici aux architectures MIMD asynchrones.

Un autre point qui différencie les ordinateurs parallèles est l'organisation de la mémoire. Elle peut être partagée par tous les processeurs ou distribuée.

4.2.1 Les architectures à mémoire partagée

Les processeurs écrivent et lisent dans le même espace d'adressage mémoire, (figure 4.1(a)). Toute opération effectuée sur la mémoire par un processeur est immédiatement visible par l'ensemble des autres processeurs. L'existence d'une mémoire partagée permet d'exploiter un parallélisme de grain fin.

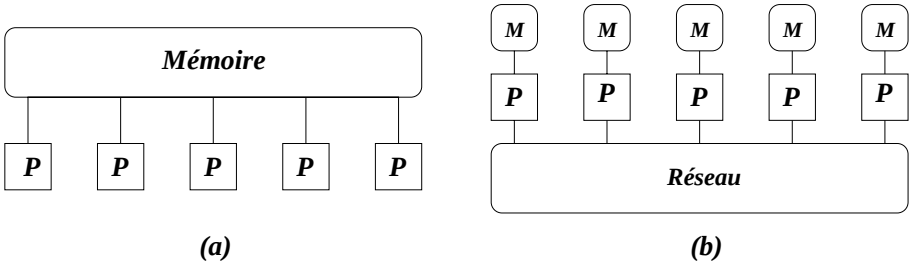

FIG. 4.1 – *(a) Machine à mémoire partagée (b) Machine à mémoire distribué*

Cependant, l'accès à la mémoire constitue un goulot d'étranglement sur ce type de machine dès que le nombre de processeurs devient important. Ce nombre se limite à une dizaine de processeurs sur les ordinateurs à mémoire partagée.

4.2.2 Les architectures à mémoire distribuée

Chaque processeur possède sa propre zone mémoire. Les différents nœuds de calcul, définis par l'ensemble $\{$processeur + mémoire$\}$, sont reliés entre eux par un réseau d'interconnexion, (figure 4.1(b)). Les nœuds de calcul communiquent entre eux par des échanges de messages à travers le réseau de communication. Le temps de communication d'un message peut être modélisé par une fonction de la forme:

Latence + taille du message/bande passante

où la latence de communication est le temps de démarrage de la communication. La bande passante du réseau est la quantité de données qui peut transiter sur un lien par unité de temps.

Les constructeurs ont essayé plusieurs topologies de réseau pour connecter les nœuds de calcul : hypercube pour le CM2 et le iPSC, grilles pour le Paragon, tore pour le Cray T3E, fat-tree pour le CM5, réseau multi-étages pour le SP1... Dans les anciennes générations de machines, la topologie du réseau est un élément critique qu'il faut prendre en considération lors de la programmation d'une application. La latence de communication entre deux nœuds peut fortement varier selon qu'ils sont voisins ou non. Il est à la charge du programmeur de découper et placer ses données de façon à ne pas dégrader les performances espérées par la parallélisation de l'application à cause des performances du réseau d'interconnexion.

La nouvelle génération de machines MIMD utilise des réseaux d'interconnexion qui assurent des latences de communication constantes entre processeurs. La plupart des machines utilisent des modes de routage dérivés du *wormhole*. Dans ce mode, acheminer un message consiste à établir un chemin dans le réseau entre le nœud source et le nœud destination. Le message est ensuite découpé en petits paquets qui sont envoyés les uns à la suite des autres sur le chemin fixé. Ce mode de routage rend les communications peu dépendantes de la topologie du réseau.

Pour améliorer les performances de communications certaines machines offrent la possibilité de recouvrir les calculs et les communications comme dans le Cray T3D et T3E et le IBM SP2. Les nœuds de calculs des machines possèdent un microprocesseur spécialisé dans la communication qui accède directement à la mémoire du nœud.

Il existe aujourd'hui des machines avec des architectures mixtes : des machines à mémoire distribuée dont les nœuds de calculs sont des ordinateurs parallèles à mémoire partagée (figure 4.2). La machine SGI-Origin 2000 en est un exemple. Ce type d'architecture est amené à se développer car il permet d'exploiter plusieurs grains de parallélisme. Notons aussi le développement des machines à mémoire partagée virtuelle, ainsi que la multiplication des réseaux de stations, que l'on appelle aussi grappe de stations, et qui présentent l'avantage d'être flexibles, modulaires et d'un coût économique relativement peu élevé.

Nous ne considérerons pour notre parallélisation que les machines parallèles MIMD à mémoire distribuée.

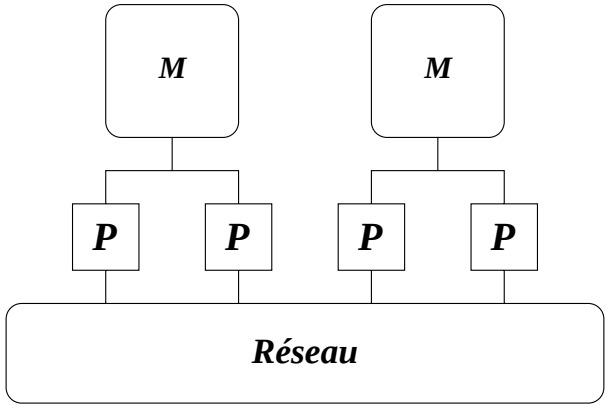

FIG. 4.2 – *Architecture mixte*

4.3 Approche de parallélisation

Dans la section précédente nous avons rappelé les architectures des machines parallèles. Dans cette section nous présenterons les différentes approches de parallélisation d'une application, ainsi que les problèmes qui se posent et les choix qui peuvent être pris durant la parallélisation. Nous ne nous intéressons qu'aux méthodes de parallélisation explicite.

Il n'existe pas d'approche type lors de la parallélisation d'une application. Ceci dépend fortement de la nature de l'application. Il est souvent nécessaire de construire un algorithme parallèle très différent de l'algorithme séquentiel. Pendant la procédure de parallélisation, il faut identifier les tâches qui peuvent s'exécuter en parallèle. Il faut ensuite choisir l'ordre d'exécution de chacune des tâches et le processeur où elles seront exécutées.

4.3.1 Sources du parallélisme

Parallélisme de données

Le parallélisme de données est la première source de parallélisme. Il consiste à effectuer la même opération en parallèle sur un ensemble de données réparties sur les processeurs. Le programme parallèle se résume en une succession de phases de calcul et de phases de communication. La phase de communication permet de mettre à jour des données distantes ou à redistribuer des données pour améliorer la localité et l'équilibrage de charge entre les processeurs.

Parallélisme de contrôle

Le parallélisme de contrôle ou fonctionnel consiste à découper une application en plusieurs tâches qui peuvent s'exécuter en parallèle. L'algorithme parallèle peut être décrit sous la forme d'un graphe orienté sans cycle de dépendance entre les tâches : une tâche ne peut être effectuée qu'après la fin de l'exécution de toutes les tâches qui la précèdent dans le graphe de dépendance. Dans ce paradigme de parallélisme, ce sont les traitements qui sont placés sur un processeur. Les données sont accédées à distance ou déplacées en tant que paramètre du traitement. Dans la suite, nous parlerons essentiellement de parallélisme de contrôle qui est plus général que le parallélisme de données.

4.3.2 Grain de parallélisme

Le grain de parallélisme peut être défini comme la longueur des séquences de calcul ou des tâches sur un nœud. Le choix de la granularité est un point essentiel pendant la description d'un algorithme parallèle. Dans le parallélisme de données, le grain de parallélisme est déterminé par la finesse de la découpe des données. Dans le parallélisme de contrôle le grain de parallélisme est déterminé par le rapport entre le temps d'exécution des tâches et la largeur du graphe des tâches, c'est à dire le nombre maximum de tâches qui peuvent s'exécuter en parallèle. On augmente le grain de parallélisme en regroupant plusieurs tâches. On obtient alors un parallélisme de gros grain. Puis on diminue la granularité en découpant les tâches en sous-tâches plus petites. On obtient alors un parallélisme de grain fin.

Le choix de granularité peut être fait en fonction des caractéristiques d'une machine cible particulière. On a déjà vu que les machines parallèles à mémoire partagée sont bien adaptées à un parallélisme de grain fin grâce à la localité des données, mais elles ne peuvent entièrement l'exploiter à cause du nombre limité de processeurs. Pour les machines à mémoire distribuée, c'est la latence de communication et la bande passante de communication qui influent sur le grain de parallélisme souhaitable. Ainsi un grain trop fin peut entraîner trop de communication entre les processeurs. Alors qu'un parallélisme de gros grain rend les calculs trop longs devant des communications négligeables et diminue la concurrence de calcul entre les processeurs ce qui entraîne un déséquilibre de charge entre les processeurs. Il est important de faire un compromis entre le grain de parallélisme et le sur-coût de la gestion du parallélisme qu'il induit. Il est parfois utile de pouvoir contrôler dynamiquement le grain de parallélisme durant l'exécution.

4.3.3 Les applications parallèles irrégulières

On peut classer les applications parallèles selon leur comportement. Une application parallèle est dite régulière si on peut prédire à l'avance le graphe des tâches et la durée des tâches. En revanche, une application parallèle est dite irrégulière si on ne peut pas prédire le comportement de l'application indépendamment de l'instance du problème qu'on cherche à résoudre. On peut avoir plusieurs causes de cette irrégularité:

- La durée de calcul ainsi que le nombre d'étapes de calculs dans une tâche ne sont pas connus et varient fortement d'une tâche à l'autre.
- On ne connait pas précisément les dépendances entre les tâches. Il est impossible de construire le graphe des tâches a priori.
- La taille des données manipulées lors des communications varie fortement ou n'est pas connue d'avance.

La parallélisation des applications irrégulières se heurte à la difficulté de prévoir même partiellement le graphe des tâches et l'estimation des coûts de calcul d'une tâche en fonction de ses entrées. Ainsi, il est difficile de bien choisir d'une part le grain de parallélisme afin d'avoir un bon équilibrage de charge, et d'autre part le placement optimal des tâches afin de minimiser les communications.

Les applications irrégulières peuvent provenir de plusieurs domaines tels que l'optimisation combinatoire, la dynamique moléculaire, la météorologie...

4.3.4 Ordonnancement

L'ordonnancement des tâches consiste à attribuer une date de début d'exécution à chaque tâche. L'ordonnancement dépend du nombre de processeurs utilisés et du graphe de précédence des tâches. Pour les applications régulières, il est "facile" d'ordonnancer les tâches puisque le graphe de précédence est connu à l'avance. Pour les applications irrégulières le graphe de précédence est connu au cours de l'exécution. On ordonnance les tâches au fur et à mesure de leur création. On utilise donc des méthodes d'ordonnancement en ligne plus compliquées à mettre en œuvre.

4.3.5 Placement et régulation de charge

L'un des points importants de la parallélisation d'une application est le choix d'un bon placement des tâches sur les processeurs minimisant les communications. Ainsi on place les tâches de manière à ce que l'accès aux données soit le plus local possible pour ne pas surcharger le réseau de communication. On place en général sur le même processeur les tâches qui ont beaucoup de données à échanger. De plus, afin de maximiser la concurrence et donc de mieux équilibrer la charge de calcul entre les processeurs, le placement doit garder le plus de tâches actives simultanément.

En pratique, le placement et l'ordonnancement de tâches s'effectuent au même moment. Pour les applications régulières on utilise des algorithmes d'ordonnancement et de placement statique. Le problème de placement peut être défini comme un problème combinatoire sur le graphe des tâches où les coûts des sommets et des arcs sont définis comme étant égaux respectivement à la durée de l'exécution de la tâche et au temps de communication des données.

Pour les applications irrégulières, on utilise des algorithmes de placement dynamique spécialement adaptés au problème. Des mécanismes de régulation de charge dynamique sont aussi intégrés à l'application. Ces algorithmes comportent plusieurs phases:

- L'évaluation de la charge de chaque processeur. Elle est en général très difficile à définir pour les applications irrégulières, puisqu'on ne connaît pas à l'avance le coût d'une tâche.
- La détermination de la surcharge ou de la sous-charge d'un processeur.
- La sélection des tâches à migrer d'un processeur à un autre.
- La migration de tâches sélectionnées.

Notons qu'un bon algorithme de régulation de charge ne doit pas augmenter considérablement les communications dues à la migration des tâches, et ne doit pas consommer beaucoup de temps de calcul durant les différentes étapes d'évaluation de la charge.

4.4 Mesure des performances

L'évaluation de performance d'un algorithme parallèle est importante pour connaître le "gain" obtenu par la parallélisation par rapport à l'algorithme sé-

quentiel.

4.4.1 Accélération

L'accélération (*Speed Up*) est définie comme le rapport du temps d'exécution séquentiel T_{seq} sur le temps de l'exécution parallèle T_p sur p processeurs:

$$S_p = \frac{T_{seq}}{T_p}$$

L'accélération idéale est égale à p. Si on considère la fonction accélération $S(p)$, l'algorithme parallèle est d'autant plus performant que la fonction $S(p)$ est proche de la droite $y = p$. On parlera dans ce cas d'accélération linéaire. Pour avoir des bonnes accélérations, il faut que le surcoût de la parallélisation en temps de calcul et surtout en volume de communication soit minimum. Il faut s'appliquer aussi à minimiser les temps d'inactivité des processeurs dûs à une mauvaise répartition de charge.

Notons qu'on peut introduire la notion d'accélération relative, par opposition à l'accélération absolue, qui ne prend pas en compte dans T_{seq} les parties intrinsèquement séqentielles de l'algorithme.

4.4.2 Efficacité

L'efficacité d'un algorithme parallèle est définie comme le rapport de l'accélération sur le nombre de processeurs.

$$E_p = \frac{S_p}{p} = \frac{T_{seq}}{p \times T_p}$$

L'efficacité représente le rendement de l'algorithme parallèle et elle est souvent exprimée par un pourcentage qui représente l'utilisation moyenne des processeurs par rapport à une utilisation permanente durant toute la durée de l'exécution T_p.

Pour approfondir toutes ces notions, le lecteur est encouragé de consulter [CT93, AGF+95].

Chapitre 5

Parallélisation du “Branch and Bound”

Dans ce chapitre nous présenterons un aperçu non exhaustif des études de parallélisation de l'algorithme du “Branch and Bound” dédié à la résolution de problème d'optimisation combinatoire. Nous rapporterons quelques exemples d'expérimentations et de plateformes de programmation de “Branch and Bound” parallèle.

5.1 L'algorithme du “Branch and Bound”

La méthode du “Branch and Bound” est l'un des plus puissants algorithmes pour résoudre des problèmes $\mathcal{NP}$-difficile. Pour la plupart des ces problèmes seulement des petites instances peuvent être résolues en un temps raisonnable sur des ordinateurs séquentiels. La parallélisation de cette méthode pour diminuer les temps de calcul a retenu l'attention de nombreux chercheurs dans les dix dernières années, en particulier pour la résolution de problèmes d'optimisation combinatoire. L'objectif de cette section est de donner un aperçu général sur l'état de ces recherches et les résultats obtenus.

Rappellons que la méthode du “Branch and Bound” consiste en une énumération implicite de l'ensemble des solutions d'un problème d'optimisation P : $\min_{x \in S} c(x)$, où $c(x)$ est une fonction coût et S est l'espace des solutions réalisables de P. Le principe de l'algorithme est de diviser l'ensemble des solutions en sous-ensembles. Cette opération s'appelle branchement. Elle consiste à construire un arbre $T = (N, E)$ où N représente l'ensemble des nœuds de l'arbre qui représentent les sous-ensembles obtenus par branchement. Ensuite un nœud est sélec-

tionné de la file des nœuds actifs puis évalué. La valeur du nœud représente une borne inférieure de la valeur de la solution optimale. Un nœud peut être élagué si son évaluation est supérieure à la meilleure solution connue, sinon le nœud est ajouté à la file des nœuds actifs. La stratégie de sélection du nœud définit la manière de parcourir l'arbre du “Branch and Bound”. Pour une bonne compréhension des phénomènes liés aux performances de l'algorithme du “Branch and Bound” Nous allons caractériser l'ensemble des nœuds de l'arbre complet d'énumération implicite. Nous reprenons les définitions de Mans et Roucairol dans [MR96] des différents sous-ensembles de nœuds inclus dans N :

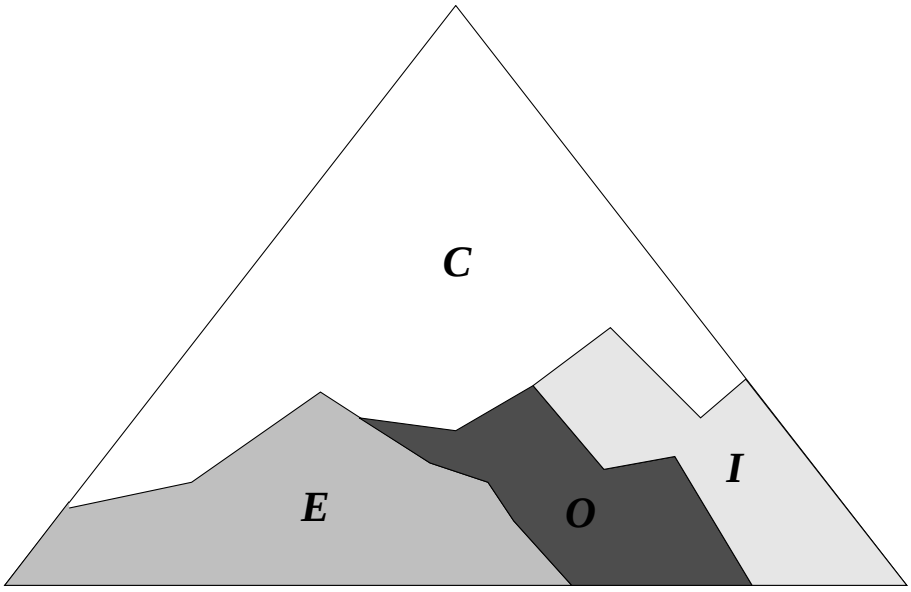

FIG. 5.1 – *Structure de l'arbre complet d'énumération du “Branch and Bound”*

– Un ensemble critique

$$C = \{N_i | c(N_i) < c(x^*)\}$$

C est l'ensemble de nœuds d'évaluation inférieure à la valeur de la solution optimale.

– Un ensemble des nœuds d'indécision

$$I = \{N_i | c(N_i) = c(x^*), N_i \text{ n'est pas réalisable}\}$$

I est l'ensemble des nœuds d'évaluation égale à la valeur de la solution optimale mais qui ne sont pas réalisables.

– Un ensemble des nœuds optimaux

$$O = \{N_i | c(N_i) = c(x^*), N_i \text{ est réalisable}\}$$

– Un ensemble des nœuds à élaguer

$$E = \{N_i | c(N_i) > c(x^*)\}$$

E est l'ensemble des nœuds d'évaluation supérieure à la valeur de la solution optimale.

L'arbre critique composé des nœuds de C, correspond à l'arbre de taille minimale que l'on doit développer pour prouver qu'une solution est optimale. L'exploration des nœuds de l'ensemble E ne peut pas conduire à une solution optimale et provoque une perte en espace mémoire et en temps de calcul. La stratégie "meilleur d'abord" interdit l'exploration des noeuds de E. Toutefois notons que la taille de l'espace mémoire requis pour stocker les noeuds actifs croît exponentiellement avec cette stratégie.

L'algorithme du "Branch and Bound" est une application irrégulière puisqu'on ne peut pas prédire son comportement indépendamment de l'instance du problème que l'on souhaite résoudre. La durée de calcul des tâches est inconnue à priori et il en est de même du nombre de tâches à traiter, ainsi que de leur dépendance.

5.2 Approche de parallélisation

On peut classer les algorithmes parallèles de "Branch and Bound" selon le grain de parallélisme utilisé. Il existe trois stratégies.

– La première consiste à introduire le parallélisme pendant l'évaluation du nœud de l'arbre du "Branch and Bound". Ce parallélisme n'influe pas sur la structure de l'algorithme du "Branch and Bound". Il dépend fortement de la méthode d'évaluation d'un sous-problème.

– La deuxième stratégie de parallélisation consiste à construire l'arbre du "Branch and Bound" en parallèle. Plusieurs nœuds de l'arbre du "Branch and Bound" sont évalués simultanément sur les processeurs de la machine parallèle.

– Le dernier type de parallélisme consiste à construire plusieurs arbres du “Branch and Bound” en parallèle avec des stratégies différentes. Sur chaque processeur, on choisit une stratégie différente de branchement, de sélection ou d’élimination... Les informations recueillies sur un processeur pendant la construction d’un arbre peuvent être utilisées sur d’autres processeurs.

Les trois stratégies de parallélisation peuvent être combinées soit d’une façon séquentielle, soit d’une façon hiérarchique. Pekny et Miller [PM92] ont parallélisé l’évaluation du noeud racine, pour passer ensuite à une construction en parallèle du reste de l’arbre du “Branch and Bound”. Mais dans la majorité une seule stratégie de parallélisation est exploitée.

5.2.1 Parallélisation de l’évaluation

La fonction d’évaluation dans un algorithme de “Branch and Bound” est spécifique au problème résolu. Pour des problèmes qui s’écrivent sous la forme d’un programme linéaire en variables mixtes, on utilise généralement, pour évaluer un nœud, un algorithme de résolution de programmes linéaires en variables réelles, comme la méthode du simplexe par exemple. Plusieurs études ont été menées pour la parallélisation des méthodes de résolution de programmes linéaires.

OSL[IBM95], la librairie de routine d’optimisation de IBM, fournit trois méthodes de résolution de programmes linéaires: La méthode de la barrière logarithmique, la méthode de décomposition par bloc et le simplexe.

La méthode de la barrière logarithmique, dite aussi méthode des points intérieures, est une méthode alternative au simplexe pour la résolution de programmes linéaires. Elle consiste à transformer un programme linéaire mis sous forme standard en reportant dans la fonction économique les bornes sur les variables. L’étape coûteuse en temps de calcul est la factorisation de Cholesky d’une certaine matrice. Il est important de réordonner les lignes et les colonnes de la matrice pour minimiser le remplissage. La parallélisation de la méthode de la barrière repose sur une décomposition par bloc de la matrice et la parallélisation des descentes-remontées. IBM annonce des accélérations de trois à cinq sur huit processeurs SP2 avec OSLp. L’usage du switch est recommandé car l’algorithme met en œuvre un parallélisme de grain fin. Les échanges sur le réseau de communication sont très importants.

Quand le programme linéaire a une structure par blocs en escalier (planning avec des couplages entre les périodes de temps) ou des structures par blocs indépendants couplées par quelques contraintes (modèles de réseau où chaque sommet

est modélisé par un programme linéaire spécifiques), OSL reconnaît automatiquement ces structures et utilise l'algorithme de Dantzig-Wolf pour la résolution. Cette méthode est parallélisable facilement. Les accélérations obtenues sont quasiment linéaires.

Malheureusement l'implémentation de l'algorithme parallèle du simplexe à grain moyen donne "des résultats désastreux sans raison apparente". Une parallélisation du simplexe avec un grain fin est actuellement à l'étude.

ILOG CPLEX fournit aussi une version parallèle des algorithmes du simplexe et de la barrière logarithmique.

La parallélisation de l'évaluation reste encore très peu exploité à cause la complexité de sa mise en œuvre et les synchronisations qu'elle nécessiterait pendant l'exécution l'algorithme du "Branch and Bound".

5.2.2 Explorations concurrentes

Cette stratégie de parallélisation utilise un gros grain. Elle est bien adaptée à des architectures MIMD asynchrones. Mais son intérêt reste très limité et donc très peu étudié. Un exemple de ce type de parallélisation est de construire des arbres de "Branch and Bound" avec des stratégies de branchement différentes [MP93].

On peut aussi faire construire les arbres de "Branch and Bound" sur chaque processeur en utilisant des stratégies de parcours différentes. Jankiram et al.[JAM88] utilisent une variante du parcours en profondeur d'abords. Cette méthode sélectionne d'une façon aléatoire le prochain nœud à évaluer parmi les derniers sous-problèmes générés. C'est une stratégie de parcours en profondeur d'abord aléatoire. La probabilité que deux processeurs construisent le même arbre du "Branch and Bound" est minime. Mais il arrive que les processeurs effectuent les même calculs particulièrement pendant le début de l'algorithme. Pour éviter cette redondance de calcul, une liste globale des statuts des sous-problèmes dans les k premiers niveaux de l'arbre du "Branch and Bound" est maintenue. Cela a permis d'obtenir de bonnes performances. Les simulations ont montré que l'algorithme se comporte beaucoup mieux sur une machine à mémoire partagée que sur une machine à mémoire distribuée.

Un autre exemple de parallélisation consiste à construire les arbres du "Branch and Bound" en parallèle avec des tests d'élagage différents [KK84]. L'idée principale dans ce genre d'algorithme est que chaque processeur effectue le test d'élimination avec une borne supérieure différente (dans le cas des problèmes de minimisation). Un processeur utilise la valeur de la meilleure borne supérieure trouvée au cours du calcul par tous les processeurs, tandis que les autres choisissent une

borne supérieure “optimiste”, c’est-à-dire de valeur égale à la valeur borne supérieure - ε, avec $\varepsilon > 0$. On peut considérer que sur de tels processeurs on construit une ε-aproximation de l’algorithme du “Branch and Bound”. Notons qu’à tout instant, il existe un processeur qui utilise $\varepsilon = 0$. Cette méthode est bien adaptée aux algorithmes dans lesquels la valeur de la borne supérieure change souvent. Ceci est particulièrement vrai lorsqu’on ne dispose pas de bonnes heuristiques de recherche de borne supérieure, principalement pour les problèmes pour lesquels on ne sait même pas si une solution réalisable existe.

5.2.3 Parallélisation de la recherche arborescente

Cette stratégie de parallélisation utilise aussi un gros grain de calcul. Elle est bien adaptée à des architectures MIMD asynchrones.

Il existe aussi quelques implémentations sur des machines massivement parallèles d’architecture SIMD de grain fin. Ces implémentations sont plutôt adaptées aux problèmes dont la fonction d’évaluation est triviale et qui s’exécute en temps constant. Dans la majorité des cas, la fonction d’évaluation est très différente d’un nœud à l’autre. Notons aussi que le surcoût dû à la synchronisation dans ce type d’implémentation est très élevé.

Nous ne nous intéresserons qu’à cette approche de parallélisation dans la suite. Nous donnerons une classification des algorithmes parallèles de ce type et les résultats théoriques et pratiques obtenus.

5.3 Classification des algorithmes de recherche arborescente parallèle

On peut classer les algorithmes parallèles de “Branch and Bound” de ce type sur machine MIMD asynchrone selon plusieurs critères. Nous avons choisi de suivre la classification proposée par Gendron et Crainic dans [GC94]:

- Algorithme synchrone avec une file d’attente unique.
- Algorithme asynchrone avec une file d’attente unique.
- Algorithme synchrone avec plusieurs files d’attente.
- Algorithme asynchrone avec plusieurs files d’attente.

Un algorithme synchrone est divisé en plusieurs phases: des phases de calcul et des phases de communication. Les communications s'effectuent entre deux phases de calculs. Tous les processeurs doivent se synchroniser avant la phase de communication. On peut distinguer deux types d'algorithmes synchrones:

- ceux dont le protocole de communication est fixé et ne varie pas d'une exécution à une autre. Dans ce dernier cas, le protocole de communication est défini par l'ensemble des informations à envoyer et leur destination; ces algorithmes ont un comportement déterministe et suivent le même cheminement d'une exécution à une autre.

- ceux dont le protocole dépend des informations recueillies pendant l'exécution et donc peuvent ne pas avoir le même comportement d'une exécution à une autre.

Dans les algorithmes asynchrones, les communications peuvent se faire à n'importe quel moment de l'exécution. Ce type d'algorithme a un comportement totalement non déterministe.

Le deuxième critère de classification est basé sur la file des nœuds de l'arbre du "Branch and Bound" en attente d'être évalués. La classification distingue les algorithmes qui possèdent une seule file et ceux qui en possèdent plusieurs.

Les algorithmes avec une file unique sont souvent implémentés sur une architecture à mémoire partagée. Dans le cas d'une architecture à mémoire distribuée, ces algorithmes sont basés sur le paradigme maître/esclave. Le processeur maître coordonne le travail entre les processeurs esclaves, et gère la file d'attente qui se trouve dans sa zone mémoire. Il envoie les nœuds de la file aux esclaves et reçoit les nouveaux nœuds générés par les esclaves. Le contrôle centralisé assure une certaine équité du travail entre les processeurs, mais nécessite l'utilisation de structures de données complexes pour réduire les goulots d'étranglement.

Les algorithmes avec plusieurs files peuvent être implémentés de différentes façons. On peut associer une file pour chaque processeur ou pour un groupe de processeurs. On peut aussi créer une file globale qui coexiste avec les files locales de chaque processeur. Ce type d'organisation est dit hybride ou mixte. Le contrôle distribué nécessite la mise en place d'algorithmes efficaces d'équilibrage de charge et de détection de la terminaison.

5.4 Equilibrage de charge dans un algorithme parallèle de “Branch and Bound” distribué

La régulation de la charge entre processeurs est un point essentiel dans n'importe quel algorithme parallèle. Le but de ce type d'algorithme est d'assurer un bon partage du travail entre les processeurs. Pendant la parallélisation du “Branch and Bound” on doit aussi prendre en compte le surcoût de la recherche arborescente engendré par la parallélisation. En effet, l'algorithme séquentiel du “Branch and Bound” avec un parcours meilleur d'abord explore à chaque étape le nœud qui possède la plus petite borne inférieure. Dans un contexte distribué, et en l'absence de communication et/ou de contrôle centralisé, la valeur de la meilleure borne inférieure n'est connue que par le processeur sur lequel le nœud est évalué. Les algorithmes distribués ont tendance donc à engendrer un surcoût dans la recherche arborescente. L'algorithme parallèle a tendance à explorer les nœuds de type E alors qu'un algorithme séquentiel avec une stratégie de parcours meilleur d'abord ne les explore pas.

Pour obtenir de bonnes performances par rapport à l'algorithme du “Branch and Bound” séquentiel trois problèmes doivent être résolus:

– Minimiser les temps morts pour chaque processeur, ce qui implique une mise en œuvre de techniques d'équilibrage de charge classiques pour assurer une équité de travail

– Minimiser le surcoût dans la recherche arborescente engendré par la parallélisation

– Minimiser les communications induites par le partage des informations et la régulation de charge.

Les buts recherchés par ces trois critères sont contradictoires. La réduction des communications implique un déséquilibrage de charge entre les processeurs et un surcoût dans la recherche arborescente. Alors que la réduction de ce surcoût implique soit l'augmentation du nombre des communications, soit un déséquilibrage de charge. Un bon algorithme de régulation de charge doit satisfaire en même temps ces trois buts pour obtenir la meilleure accélération.

Les algorithmes de régulation de charge peuvent être statiques ou dynamiques. Les algorithmes statiques placent initialement les tâches sur les processeurs afin de minimiser les communications et d'assurer un équilibre de charge entre les

processeurs. Si la charge change dynamiquement pendant l'exécution d'une façon irrégulière, il est alors nécessaire d'utiliser une régulation dynamique de charge qui réagit à ces changements.

L'équilibrage de charge peut être effectué d'une manière centralisée suivant un paradigme Maître/Esclave. Le processeur maître reçoit les requêtes des esclaves et décide à quel processeur esclave il doit envoyer du travail. Ce modèle assure une parfaite équité du travail et n'induit aucun surcoût dans la recherche arborescente. Toutefois il induit un surcoût de communications et un goulot d'étranglement autour du processeur maître. Pour remédier à ce problème on peut adopter un modèle hiérarchique ou un modèle complètement distribué.

Dans un modèle distribué, on distingue trois modes selon le type de processeur qui prend la décision de l'équilibrage.

- Mode passif : Dans ce mode, les processeurs inactifs initient la demande de répartition de charge. Ce mode peut induire un surcoût de communication lors de la phase initiale et finale de l'algorithme du “Branch and Bound”. Les multiplications des requêtes pendant ces phases peuvent gêner les processeurs actifs.

- Mode actif : Dans ce mode, les processeurs actifs saturés proposent de partager leurs charges. Dans ce mode, si les processeurs actifs ne sont pas saturés, ceux qui sont inactifs ne reçoivent jamais de travail. Ce mode est intéressant pendant la phase initiale et finale de l'algorithme du “Branch and Bound”.

- Mode mixte : Ce mode est un mélange des deux autres.

Il existe plusieurs stratégies d'équilibrage de charge dans un contexte complètement distribué. Nous expliquons les principales stratégies utilisées dans les expérimentations présentées dans les sections suivantes.

5.4.1 Stratégie de l'ardoise

Dans cette stratégie, on utilise une zone mémoire partagée par tous les processeurs, nommée ardoise. Après la sélection d'un nœud de sa file locale, le processeur ne le traite que s'il est d'assez bonne qualité par rapport au meilleur nœud sur l'ardoise. C'est la valeur de l'évalution d'un nœud qui détermine sa qualité par rapport aux autres. Ainsi dans un algorithme de “Branch and Bound” utilisant la stratégie de parcours meilleur d'abord, les nœuds intéressants sont les nœuds d'évaluation minimum, quand on traite un problème de minimisation. Donc, si le

nœud sélectionné par un processeur est de meilleur qualité que le meilleur nœud de l'ardoise, alors le processeur sélectionne quelques nœuds de bonne qualité de sa file d'attente et les transfère sur l'ardoise. Par contre, si le nœud sélectionné a une borne très mauvaise par rapport au meilleur nœud de l'ardoise alors le processeur récupère quelques nœuds de l'ardoise pour les placer dans sa file locale.

Le choix du seuil de tolérance de la différence entre la meilleure borne de la file locale et de l'ardoise est très important, puisqu'il induit le nombre de nœuds échangés entre l'ardoise et les files locales. Si ce seuil est très bas alors le nombre de nœuds échangés est très grand, à moins que les bornes soient toutes assez proches. Si le seuil de tolérance est assez grand alors le nombre de nœuds échangés devient moins important, ce qui peut causer un surcoût dans la recherche arborescente.

Ce type de stratégie reste limité aux machines possèdant un nombre restreint de processeurs, à cause des problèmes de congestion autour de la mémoire partagée.

5.4.2 Stratégie aléatoire

Dans cette stratégie, périodiquement chaque processeur choisit aléatoirement un autre processeur pour lui envoyer un de ses meilleurs nœuds. Ceci assure un une bonne répartition de la charge de calcul et un bon partage des nœuds de bonne qualité à traiter. Cette stratégie est facile à implémenter. Si les transferts sont assez fréquents alors le surcoût dans la recherche arborescente est minime mais le réseau est très vite saturé. Dans le cas contraire, on observe des surcoûts importants. Le choix de la périodicité est déterminé par rapport au coût des communications. Notons que cette stratégie peut être appliquée à une topologie virtuelle du réseau de communication, dans ce cas un processeur choisit aléatoirement un de ses voisins dans cette topologie.

5.4.3 Stratégie des seuils

Dans ce type de stratégie, on introduit une fonction poids w qui définit la charge de chaque processeur. Soit $p_i \in \{p_1, \ldots, p_n\}$ un processeur et bsg la valeur de la meilleure solution réalisable (borne supérieure), bil la borne inférieure d'un nœud de l'arbre du “Branch and Bound” et $F_{p_i} = \{n_1, \ldots, n_t\}$ la file des nœuds en attente sur le processeur p_i. On définit les fonctions poids comme suit:

$$w_{bil}(p_i) = min\{bil(n_i)|n_i \in F_{p_i}\}$$

$$w_{\#}(p_i) = |F_{p_i}|$$

Le but est d'optimiser l'équilibrage de charge selon les deux critères définis par la qualité des nœuds traités w_{bil} et la charge de calcul $w_{\#}$. Quand on essaie de garantir à un instant donné le même niveau de la borne inférieure sur tous les processeurs, la charge de calcul peut être déséquilibrée. Des processeurs peuvent se retrouver sans assez de place mémoire pour effectuer leur calcul alors que d'autres en disposent de beaucoup. D'où le besoin d'introduire une seconde fonction poids $w_{\#}$ qui donne le nombre de nœuds en attente dans la file de chaque processeur. Il est possible de définir une seule fonction poids pour les deux critères.

Dans cette stratégie, on essaie de garder tous les processeurs au même niveau selon les deux poids . Pour ce faire, chaque processeur connaît les poids $w_{\#}$ et w_{bil} de ses voisins. Un processeur envoie quelques nœuds si la qualité ou la quantité de sa file est supérieure d'un certain niveau à celle de ses voisins. Si la situation de la file locale change, les voisins sont informés. Avec cette stratégie on atteint un bon équilibrage tout au long de l'exécution avec un surcoût minimal dans la recherche arborescente. Pour bien définir la notion de voisinage utilisée dans cette stratégie, on construit une topologie virtuelle du réseau.

5.5 Les anomalies de comportement des algorithmes parallèles de “Branch and Bound”

Il existe des études théoriques sur le comportement des algorithmes de “Branch and Bound” synchrones à file d'attente unique. Dans ce type d'algorithmes les sous-problèmes sont stockés dans une file d'attente unique. La borne supérieure est stockée dans une variable globale accessible par tous les processeurs. Initialement le nœud racine est ajouté à la file F. A chaque itération $min(|F|, p)$ nœuds de F sont choisis selon un certain critère de sélection. Chaque nœud sélectionné est traité par un processeur. Chaque processeur effectue l'opération de branchement, d'évaluation et le test d'élimination. A la fin de chaque itération la borne supérieure est mise à jour et les nouveaux nœuds générés sont ajoutés à la file F. L'algorithme s'arrête quand la file F est vide au début d'une itération.

Les stratégies de sélection peuvent être caractérisées par une fonction qui associe à chaque nœud une valeur. On sélectionne le nœud qui possède la plus petite valeur. La stratégie “meilleure d’abord” consiste à choisir le sommet qui possède la plus petite borne inférieure. Dans ce cas la valeur d’un nœud est égale à la valeur de sa borne inférieure. Cette même fonction rend l’opposé de la profondeur d’un nœud quand le critère de sélection suit la stratégie profondeur d’abord.

La fonction de sélection joue un rôle central dans la compréhension des anomalies de comportement de certains algorithmes parallèles de “Branch and Bound”. On dégage deux types d’anomalies:

- Anomalie d’accélération: $S_p > p$ où p est le nombre de processeurs et S_p l’accélération associée à p.

- Anomalie préjudiciable de croissance: $T_{p_1}/T_{p_2} < 1$ avec $p_1 < p_2$ où T_{p_1} est le temps d’exécution sur p_1 processeurs, et T_{p_2} est le temps d’exécution sur p_2 processeurs.

Les anomalies de comportement des algorithmes parallèles de “Branch and Bound” utilisant la stratégie de sélection “meilleur d’abord” ont été étudiées par Lai et Sahni [LS84] et Lai et Sprague [LS85]. Ils ont montré que les anomalies d’accélération ne peuvent pas se produire si les valeurs de tous les nœuds internes à l’arbre du “Branch and Bound” sont différentes de la valeur de la solution optimale. Mans et Roucairol [MR96] ont proposé des règles de gestion des nœuds de mêmes priorités pour éliminer les anomalies. Quin et Deo [QD85] ont donné des bornes supérieures de l’accélération des algorithmes asynchrones à file d’attente unique utilisant le critère de sélection “meilleur d’abord”. Corrêa et Ferreira ont presenté un état de l’art de ses anomalies dans [CF95b].

5.6 Exemples d’expérimentations

Roucairol [Rou87]

Roucairol a implémenté un algorithme asynchrone à file unique avec une stratégie “meilleur d’abord” sur le Cray X-MP 48. C’est une machine à mémoire partagée avec 4 processeurs. Les accélérations observées pendant la résolution du problème d’affectation quadratique sont quasiment linéaires.

Kumar, Ramesh et Nageshwara Rao [KRNR88]

Kumar, Ramesh et Nageshwara Rao ont étudié les performances de différents algorithmes parallèles de "Branch and Bound" sur le problème du recouvrement d'un graphe par les sommets et le problème du voyageur de commerce.

Ils ont implémenté un algorithme centralisé similaire à celui de Roucairol mais en utilisant une structure de données de tas simple ou concurrents pour la file d'attente. Cette dernière structure permet d'effectuer d'une façon concurrente des insertions et des éliminations. Les résultats sur une machine BBN Butterfly à 100 processeurs montrent que l'utilisation des structures de données concurrentes améliore considérablement les performances de l'algorithme pour le problème du voyageur de commerce. Ils obtiennent des accélérations quasiment linéaires pour une instance de 25 villes. Le contrôle centralisé reste une source de goulot d'étranglement lors de l'accès à la file des nœuds en attente.

Dans leur implémentation distribuée, chaque processeur possède une partie de la file des nœuds en attente. Cette stratégie exclut les goulots d'étranglement mais augmente le nombre de nœuds à visiter, puisque quelques processeurs peuvent posséder tous les nœuds de bonne qualité, alors que d'autres n'ont que des nœuds non prometteurs. Ils ont essayé trois politiques de distribution des bons candidats : l'ardoise, l'anneau et la distribution aléatoire. Leurs résultats donnent de très mauvaises accélérations pour les deux dernières stratégies de communication. Ils montrent que ces deux stratégies ne peuvent bien fonctionner que s'il existe plusieurs nœuds de l'arbre du "Branch and Bound" de valeurs égales ou très proches. La stratégie de l'ardoise reste limitée par l'extensibilité de l'accès à une mémoire partagée qui est en plus très difficile à construire.

Miller et Pekny [MP89]

Miller et Pekny proposent un algorithme de résolution du voyageur de commerce assymétrique sur la machine BBN Butterfly à 14 processeurs. Ils utilisent un modèle asynchrone centralisé. Mais dans leur implémentation la file d'attente est composée de deux listes: une liste des nœuds non évalués et une liste des nœuds évalués mais non encore branchés. Pendant la sélection, un nœud non évalué est prioritaire sur un nœud évalué mais non branché. Une heuristique est utilisée pour trouver de temps en temps une solution réalisable du problème pour améliorer la valeur de la borne supérieure. Ils ont résolu des instances qui possèdent jusqu'à 3000 villes générées aléatoirement, mais il est bien connu que de telles instances sont en général faciles.

Quinn [Qui90]

Quinn [Qui90] a implémenté un algorithme synchrone à file unique sur une machine NCUBE/7 hypercube avec 64 processeurs. Les nœuds de l'arbre du “Branch and Bound” sont stockés dans une file d'attente unique dans la mémoire du processeur maître. Ce dernier gère l'affectation des sous-problèmes aux esclaves selon une stratégie “meilleur d'abord”. Des expérimentations sont rapportées sur des instances du problème du voyageur de commerce jusqu'à 30 villes. Un modèle qui décrit les performances de ces algorithmes est proposé. Ce modèle prédit une éventuelle baisse de l'accélération due au surcoût de communication. Une implémentation et un modèle d'un algorithme asynchrone à file distribuée sont aussi proposés.

Tschöke, Lüling et Monien [TLM95]

Tschöke, Lüling et Monien ont implémenté un algorithme parallèle à contrôle distribué pour résoudre le problème du voyageur de commerce. Ils ont réussi à résoudre des instances de TSPLIB [Rei91] qui possèdent jusqu'à 318 villes sur un réseau de 1024 transputers configurable. Leur algorithme utilise une stratégie de parcours “meilleur d'abord”. Ils utilisent une méthode d'équilibrage de charge dynamique qui tente de maintenir tout au long de l'exécution les processeurs à un même niveau de charge. Le niveau de charge est défini par deux coûts : la valeur de la plus petite borne inférieure de la file d'un processeur et le nombre de nœuds dans la file. Chaque processeur connaît la valeur de ses voisins dans le réseau. Un processeur envoie des sous-problèmes à son voisin, si la différence de qualité ou de quantité des nœuds entre les deux files dépasse un certain seuil. Si les coûts associés à sa file changent, il informe ses voisins. Cette stratégie d'équilibrage s'est avérée très efficace surtout avec un réseau à faible diamètre comme pour une topologie de De Bruijn ou une topologie en tore.

Corrêra et Ferreira [CF95a]

Corrêra et Ferreira ont implémenté un algorithme de “Branch and Bound” parallèle pour résoudre le problème du sac à dos sur un Intel Paragon avec 32 processeurs. Ils ont proposé une version synchrone qui donne un très bon partage de charges entre les processeurs, mais des accélérations moyennes à cause de la fine granularité de l'application. Ils ont aussi proposé une version asynchrone et flexible de leur algorithme, celle-ci donne de très bonnes accélérations sur 20 instances du sac à dos de taille 180 générées aléatoirement. L'algorithme consiste

à effectuer une parallélisation du parcours de l’arbre du “Branch and Bound” selon une stratégie “meilleure d’abord”.

Denneulin, Le Cun, Mautor et Méhaut [DLCMM96]

Denneulin, Le Cun, Mautor et Méhaut ont implémenté un algorithme parallèle de “Branch and Bound” pour résoudre le problème d’affectation quadratique sur un environnement distribué. Deux approches ont été utilisées. La première est basée sur un modèle maître/esclave. Elle donne des accélérations quasiment linéaires sur un DEC ALPHA avec 16 processeurs ou sur le IBM-SP2. Ces performances se détériorent dès que le nombre de processeurs augmente. La deuxième approche est complètement distribuée et donne de très bonnes performances. L’équilibrage de charge utilisé ne tient pas compte de la qualité des nœuds placés sur les processeurs. Des résultats sont donnés pour des instances de QAPLIB [BKR94] de taille 16 et 20.

5.7 Plate-forme de programmation de “Branch and Bound” parallèle

Les bibliothèques pour la résolution des problèmes avec un “Branch and Bound” parallèle sont apparue ces dernieres années. Elles fournissent les composantes principales de l’algorithme telle que la gestion de la file d’attente, de la bornne supérieure, le parallélisme...

Nous présentons ici trois bibliothèques BOB, ZRAM et PUBB.

5.7.1 BOB : “Une plate-forme Unifiée de Développement pour les Algorithmes de type Branch and Bound”

Benaïchouche, Le Cun, Dowaji, Mautor et Roucairol [Rou96, LCR95]

D’aprés les concepteurs de la plate-forme: “Le but de la bibliothèque BOB est de fournir un ensemble de fonctionnalités et de variables globales qui soient indépendantes de l’application et communes à tous les algorithmes “Branch and Bound”.

Les fonctions de la bibliothèque s’occupent d’une façon transparente de:

– La gestion du lancement des processeurs (distribués ou parallèles) selon le type de la machine cible.

– La gestion de files de priorités fondée sur la notion importante de file de priorité globale permettant l’abstraction de l’architecture des machines cibles vis-à-vis de l’algorithme “Branch and Bound”; les files de priorités peuvent être parallèles, concurrentes ou séquentielles, avec les différentes stratégies d’équilibrage de la charge s’il y a lieu.

– La gestion de la borne supérieure.

Ces différentes gestions sont rendues transparentes à un utilisateur de la bibliothèque, par l’appel à des fonctions prédéfinies.

Le type de la machine peut être reconnu automatiquement. Pour une même application, plusieurs exécutables peuvent être générés. Ils peuvent être purement séquentiels, distribué ou parallèles. Chacun d’eux peut utiliser différentes files de priorités. Les versions distribuées ou parallèles peuvent utiliser différentes stratégies d’équilibrage de charge. Des options de compilation du fichier makefile définissent donc le modèle d’exécution, la file de priorité (D-Heap, Skew-Heap, Splay-Trees, etc.), les stratégies d’équilibrage de charge.

L’utilisateur doit fournir à BOB quatre fonctions décrivant sa méthode de résolution. Toutes les autres fonctions nécessaires à l’algorithme du “Branch and Bound” sont générées par le noyau de la bibliothèque. L’utilisateur n’a pas besoin de bien connaître les fonctions du noyau si ce n’est pas pour enrichir la bibliothèque en files de priorités, stratégies d’équilibrages de charges et méthodes de parallélisation. Le moniteur a pour but principal de fournir des statistiques sur les différents objets manipulés par une application.”

5.7.2 ZRAM : “A Workbench and Program Library for Portable Parallel Search Algorithms”

Brüngger, Marzetta, Fukuda et Nievergelt [BMFN96]

ZRAM est une bibliothèque parallèle portable d’algorithme de recherche exhaustive. Elle est basée sur trois interfaces qui séparent quatre couches logiciels, figure 5.2:

– *Interface de passage de messages*: cette interface cache l’architecture de la machine cible. ZRAM est conçue autour d’un sous-ensemble de l’interface standard de passage de message MPI [DA95]. Elle n’utilise que les primitives de communication d’envoi et de réception point à point non bloquante. Cette interface de passage de message a été implémentée pour des machines

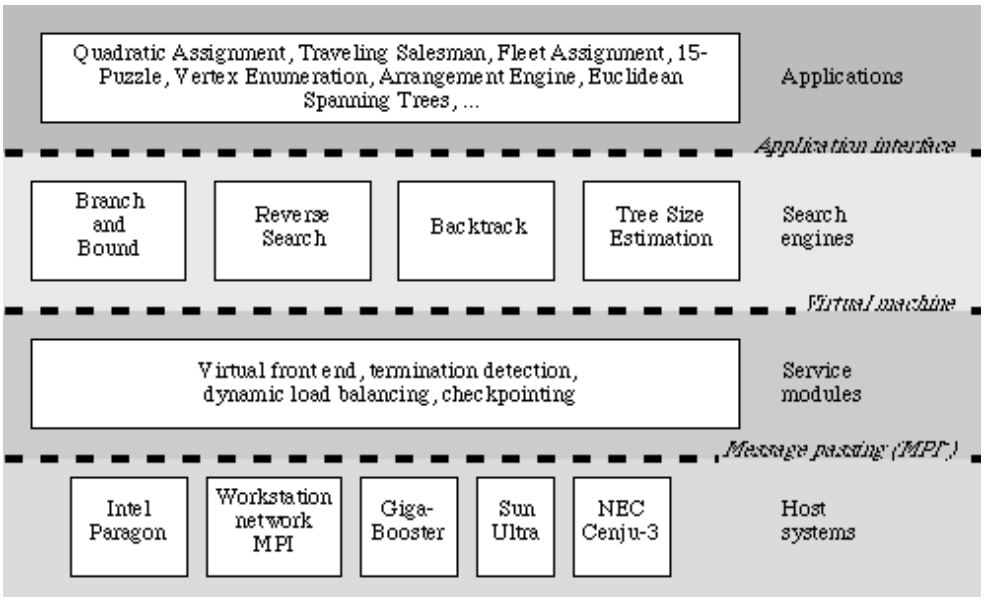

FIG. 5.2 – *Structure de ZRAM*

telles que Intel Paragon, NEC Cenju-3, ETH Giga Booster et un réseau de stations de travail utilisant MPI.

– *La machine virtuelle*: cette couche contient toutes les fonctionnalités communes à plusieurs programmes parallèles de recherche arborescente: équilibrage de charge dynamique, détection de terminaison et contrôle de l’exécution. Le calcul peut être interrompu puis repris avec un nombre différent de processeurs.

– *L’interface applicative*: cette couche implémente les algorithmes de recherche que supportent ZRAM, tel que le “Branch and Bound”, le *Backtrack*, la recherche inversée et l’estimation de la taille de l’arbre.

– *Les applications*: cette couche contient les applications programmées par l’utilisateur de ZRAM. Elle ne contient que les fonctions spécifiques à l’application. Il n’existe pas de parallélisme explicite dans cette couche.

Plusieurs applications ont été développées sur cette plate-forme telles que le problème d’affectation quadratique, l’énumération de sommets d’un polyèdre, le taquin de taille 15. Brüngger, Marzetta, Clausen et Perregaard [BMCP97, BMCP98]

rapportent une implémentation du problème d'affectation quadratique utilisant ZRAM sur l'Intel Paragon XP/S22 avec 148 processeurs et le NEC Cenju-3 avec 128 processeurs. Ils ont résolu 10 nouvelles instances de QAPLIB. La plus grande instance résolue est nug22. Son arbre du “Branch and Bound” associé possède 5.10^{10} nœuds. La solution a été trouvée en à peu près 12 jours sur un nombre de processeurs variant au cours de l'exécution entre 48 et 96 sur le NEC Cenju-3.

5.7.3 PUBB : “Parallelization Utility for Branch and Bound algorithms”

Shinano, Harada et Hirabayashi [SHH97]

Cette plate-forme est codée en C++ et utilise PVM (Parallel Virtuel Machine) [GBD+95]. PUBB possède trois modes d'exécution selon la stratégie de contrôle de la recherche arborescente: un mode de contrôle centralisé type Maître/Esclave, un mode distribué, et un mode mixte. La plate-forme est composée par trois entités, figure 5.3:

- *Problem Manager* (PM): Il maintient durant l'exécution les données du problème et la solution. Dans le mode Maître/Esclave, il maintient la liste des sous problèmes, et une fois initialisé, il se comporte comme un serveur gérant les requêtes des autres processeurs.
- *Load Balancer* (LB): Il s'occupe de l'équilibrage de charge en coopération avec les autres LBs.
- *Solver*: Les sous problèmes sont évalués dans le solveur. Dans le mode distribué, il possède une liste des sous problèmes. Un solveur ne communique qu'avec le LB qui lui est associé. Ils s'exécutent sur le même processeur.

La résolution du problème du voyageur de commerce a été programmée avec PUBB sur une grappe de machines. Des résultats sont rapportés pour des instances de TSPLIB avec les trois modes de contrôle. Cette implémentation montre l'efficacité du contrôle mixte de la recherche arborescente.

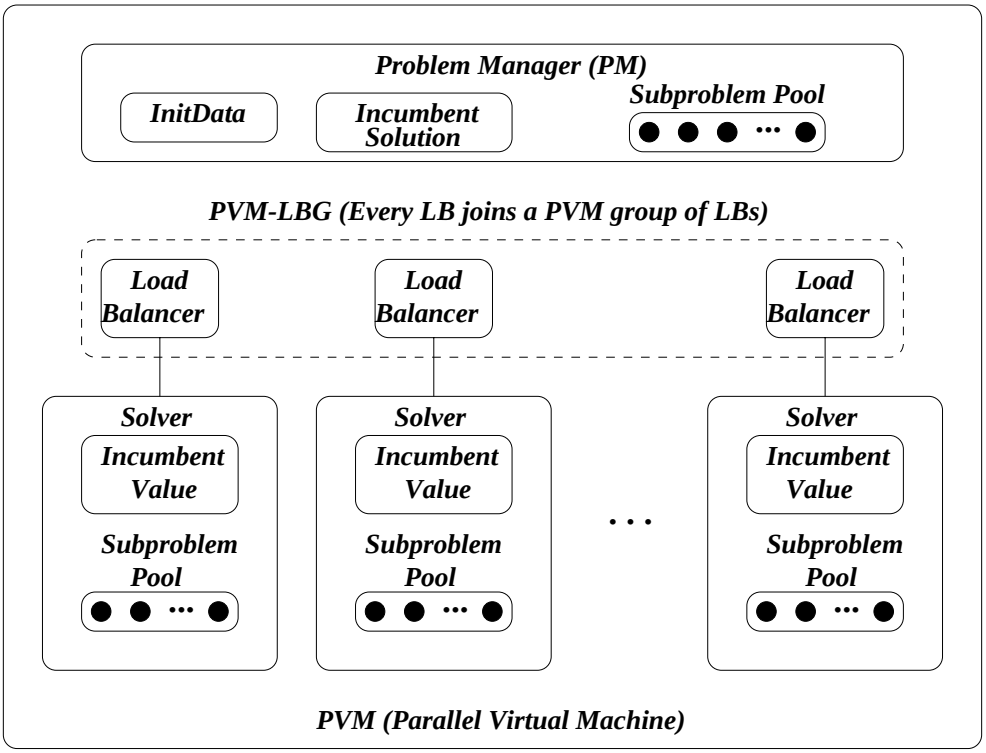

FIG. 5.3 – *Structure de PUBB*

Chapitre 6

Parallélisation du “Branch and Cut”

Dans ce chapitre nous présenterons notre méthodologie de parallélisation de l’algorithme du “Branch and Cut”. Nous commencerons par étudier les différents niveaux de parallélisme possibles. Nous donnerons ensuite la stratégie de contrôle de la recherche arborescente qu’on a adoptée. Nous montrerons enfin les mécanismes de contrôle de communication et d’équilibrage de charge et justifierons nos choix.

6.1 Niveau de parallélisme

Comme pour l’algorithme du “Branch and Bound”, l’algorithme du “Branch and Cut” se présente comme un ensemble de tâches avec très peu de dépendances de données, voire souvent aucune. Ceci implique que les processus qui effectuent des calculs sur ces données ont peu de dépendances entre eux et peuvent être ordonnancés dans un ordre quelconque. La seule relation pouvant exister entre deux tâches étant celle de parenté. Ainsi le graphe de dépendance est un arbre qui est peu exploitable. Les temps de calculs des nœuds de l’arbre ne peuvent pas être évalués a priori. Ceci est d’autant plus vrai dans un algorithme de “Branch and Cut” que l’évaluation d’un nœud utilise les méthodes de résolution de programme linéaire, de coupes polyédrales et de génération de colonnes.

Il n’est donc pas facile de paralléliser efficacement une telle application puisque son irrégularité empêche de prédire la forme de l’arbre, sa taille et la quantité de travail que représente chaque nœud dans l’arbre.

Nous avons étudié plusieurs stratégies possibles de parallélisation du “Branch and Cut”. Chaque stratégie correspond à un niveau de granularité du parallélisme. Nous présenterons trois niveaux de parallélisation: une parallélisation avec un gros

grain, une autre avec un grain moyen et une dernière avec un grain fin. Notons que la notion de fine granularité ici, est relative et n’a rien à voir avec l’ordre de grandeur utilisé dans les applications de type analyse numérique.

6.1.1 Parallélisation de gros grain: Explorations concurrentes

La première stratégie de parallélisation consiste à exécuter plusieurs “Branch and Cut” en parallèle, c’est à dire effectuer des explorations concurrentes. Dans chaque exécution, on prend des choix différents de parcours, d’initialisation ou de méthode de résolution. En échangeant des informations entre les exécutions cela nous permet d’accélérer une ou plusieurs explorations. Dans cette stratégie, dès que l’une des exécutions est finie, on arrête toutes les autres puisque à la fin d’une exécution d’un algorithme de “Branch and Cut”, on est sûr que la meilleure solution réalisable trouvée au cours du calcul est la solution optimale du problème.

Dans l’algorithme du “Branch and Cut”, on initialise une exploration par un choix arbitraire de variables de départ défini par les arêtes d’un graphe partiel initial. Dans le cas d’explorations concurrentes, chaque processus peut commencer par un graphe partiel différent. Au fur et à mesure de l’exploration, chaque exécution fait entrer ou sortir des variables dans l’ensemble des variables actives soit avec la méthode de génération de colonnes, soit en les fixant à 1 ou à 0. Quelques variables être fixées pour tous les nœuds de l’arbre. Il est donc possible de partager ces informations avec tous les autres processus. Quand un processus reçoit de nouvelles variables fixées, il peut élaguer toutes les branches où les variables reçues sont mises à une valeur contradictoire.

Nous avons montré que dans l’algorithme séquentiel, nous pouvons utiliser plusieurs heuristiques de recherches de contraintes violées avec des priorités différentes. Quelques heuristiques ne sont utilisées que si d’autres ont échoué à trouver des contraintes violées. Dans une parallélisation de gros grain on peut utiliser des heuristiques différentes de recherches de contraintes sur chaque processus. Ceci pourrait permettre de trouver une variété de contraintes qu’on n’aurait pas pu trouver dans une exécution séquentielle soit faute de temps, soit parce que l’une des solutions fractionnaires d’une exploration n’était pas bien adaptée à une heuristique. La parallélisation exploite cette variété, en échangeant des contraintes de temps en temps entre les processus. Les contraintes échangées sont choisies parmi les contraintes les plus difficiles à générer ou qui présentent une violation maximum. Ces contraintes sont stockées dans le pool de chaque processus.

Comme dans le “Branch and Bound”, chaque exploration peut s’effectuer avec une stratégie de branchement différente ou avec une stratégie différente de par-

cours de l'arbre du "Branch and Cut".

Dans ce type de parallélisation, les échanges qui peuvent avoir une influence significative sur l'exécution sont les variables fixées et la valeur de la borne supérieure. Elles permettent d'élaguer rapidement plusieurs branches de l'arbre du "Branch and Cut" et donc d'accélérer l'exécution de l'algorithme. Malheureusement la fréquence de fixations des variables n'est pas très élevée surtout au début de l'algorithme. De même on n'améliore pas fréquemment la valeur de la borne supérieure.

Une parallélisation de gros grain reste d'intérêt limité puisqu'elle ne permet pas d'accélérer considérablement le temps de calcul, et avoir une bonne efficacité de parallélisation. D'autre part, elle ne permet pas un partage des données sur les différents processus, et donc une utilisation optimisée de la mémoire pour pouvoir résoudre de plus grandes instances du problème du voyageur de commerce

6.1.2 Parallélisation de grain moyen: Parallélisation du parcours

Nous avons vu que l'algorithme du "Branch and Cut" se présente comme un ensemble de tâches indépendantes. Ainsi dans une parallélisation de grain moyen, chaque processus peut explorer une partie de l'espace de recherche. C'est l'approche la plus adaptée aux architectures parallèles que nous visons, et que nous avons retenu pour paralléliser la résolution du problème du voyageur de commerce par l'algorithme du "Branch and Cut".

Dans les sections suivantes, nous détaillerons notre implémentation de la recherche arborescente, qui prend en compte les spécificités de l'algorithme du "Branch and Cut" par rapport à l'algorithme du "Branch and Bound".

6.1.3 Parallélisation de grain fin: Parallélisation de l'évaluation

Dans ce niveau, l'évaluation de chaque nœud de l'arbre du "Branch and Cut" se fait en parallèle par plusieurs processus. Cette approche n'est viable que si cette opération est très coûteuse au regard de la gestion et du parcours de l'arbre.

L'évaluation d'un nœud de l'arbre du "Branch and Cut" est composée de plusieurs étapes:

– la résolution du programme linéaire.

– la recherche de contraintes violées.

– le “pricing” et la fixation des variables.

– la recherche de la borne supérieure.

– la recherche de la contrainte ou de la variable de branchement.

Certaines de ces étapes peuvent s’effectuer en parallèle pour un nœud donné. L’expérience nous montre que le gain escompté de la parallélisation de l’évaluation d’un seul nœud à la fois ne conduit pas nécessairement à une amélioration des temps de calcul. Ceci est dû d’une part, aux synchronisations nécessaires des processus à la fin de l’évaluation d’un nœud, ce qui ralentit considérablement l’exécution. Et d’autre part, à l’existence d’un goulot d’étranglement autour du processus qui résout le programme linéaire. En effet, la résolution du programme linéaire est la partie qui consomme le plus de temps de calcul, en particulier pour les instances de grande taille. La génération fréquente de nouvelles contraintes, l’addition et la fixation de variables est aussi un facteur de ralentissement puisqu’ils induisent des communications fréquentes entre le processus qui résout le programme linéaire et les autres.

Pour obtenir une parallélisation efficace de grain fin, il est nécessaire de mettre en œuvre une parallélisation de l’évaluation de plusieurs nœuds à la fois. Il est possible ainsi de diminuer les goulots d’étranglement et augmenter la concurrence des tâches. Les tâches considérées sont les parties indépendantes de l’évaluation d’un nœud de l’arbre: résolution du programme linéaire, recherche des contraintes violées, fixation des variables, pricing, recherche de la contrainte ou variables de branchement. Chaque opération peut être exécutée sur un processus. Chaque processus peut effectuer n’importe quel type de tâche de n’importe quel nœud.

Après l’exécution d’une tâche, le processus crée éventuellement une nouvelle tâche avec un identificateur contenant le type de tâche et le numéro de son nœud. L’identificateur de chaque tâche permet au processus d’ordonnancement de déterminer sa priorité par rapport aux autres et ainsi d’ordonnancer les tâches selon leurs priorités et leurs contraintes de précédences dans une file d’attente. Un processus de placement s’occupe du placement des tâches en respectant l’ordre des tâches dans la file d’attente tout en assurant une certaine équité de la charge de travail entre les processus.

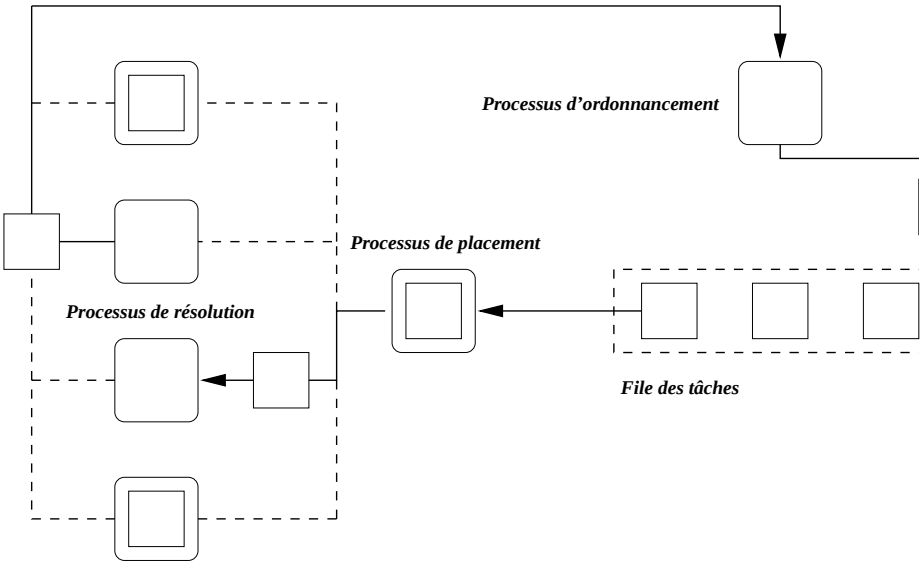

FIG. 6.1 – *Structure de l'algorithme de parallélisation de l'évalution*

La figure 6.1 donne la structure de cet algorithme. Elle contient cinq types de tâches:

– Type 1: Ce type de tâche s'occupe de la résolution du programme linéaire et de la séparation des contraintes de sous tour. A la fin de cette tâche on obtient une solution optimale du programme linéaire courant d'un nœud λ de l'arbre qui ne viole aucune contrainte de sous-tour. Cette tâche peut générer deux types de tâches, soit une tâche de recherche de contraintes violées, soit une tâche de "pricing" et de fixation des variables.

– Type 2: Ce type de tâche recherche des contraintes violées par les heuristiques de séparation et/ou dans le pool de contraintes. Si des contraintes sont générées alors elle ajoute les nouvelles contraintes au programme linéaire et crée une tâche de type 1. Sinon elle crée une tâche de pricing. Notons qu'il est possible d'exécuter cette tâche en parallèle si le nombre de processus disponible est élévé.

– Type 3: Ce type de tâche effectue le "pricing" et éventuellement des fixations de variables. Elle génère ensuite soit une tâche de type 1, soit une tâche branchement.

– Type 4: Cette tâche recherche la contrainte ou la variable de branchement. Elle génère deux tâches de type 1 avec un nouvel identificateur de nœud pour chacune ($2 \times \lambda + 1$) et ($2 \times \lambda + 2$).

– Type 5: Ce type de tâche effectue des opérations auxiliaires telles que la recherche de la borne supérieure, ou la réduction de la taille du pool de contraintes. Ce type de tâche est généré par le processus maître de temps en temps en fonction de la charge des processus. Ce type de tâche ne génère pas de nouvelle tâche.

Notons que les tâches de type 2 s’occupent aussi de la réduction de la taille du programme linéaire en enlevant les contraintes lâches. Quand la borne supérieure change, il est possible d’élaguer tous les nœuds dont la valeur de la borne inférieure dépasse la valeur de la borne supérieure. Pour cela on peut élaguer toutes les tâches d’un nœud de la file d’attente, à condition qu’on soit sûr que la valeur de la borne inférieure est valide sur le graphe complet. Pour le faire, il suffit d’effecteur une opération de pricing sur la dernière tâche de type 1 du nœud à élaguer.

Il existe aussi des travaux sur la parallélisation de certaines tâches de l’évaluation d’un nœud de l’arbre du “Branch and Cut”. T. Christof et G. Reinelt [CR95] se sont intéressé à la parallélisation de la recherche des contraintes violées en affectant à chaque processeurs la séparation d’un type de facettes du $STSP(n)$.

6.2 Contrôle de la recherche arborescente

Dans une parallélisation du parcours de l’arbre du “Branch and Cut”, le contrôle de la recherche arborescente peut se faire selon un mode centralisé avec un paradigme maître esclave ou selon un mode distribué, ces modes ont été utilisés lors de la parallélisation de l’algorithme du “Branch and Bound”. Le mode centralisé se comporte assez bien mais perd de son efficacité quand le nombre du processus augmente considérablement. Ceci est dû aux goulots d’étranglement autour du processus maître.

La philosophie de l’algorithme du “Branch and Cut” diffère de celle du “Branch and Bound”. En effet, l’évaluation d’un nœud de l’arbre du “Branch and Cut” est plus lente puisqu’elle conjugue plusieurs algorithmes. Ainsi, l’effort de calcul de la borne inférieure est beaucoup plus important que dans l’algorithme du “Branch and Bound” et la taille de l’arbre est beaucoup plus petite. Les plus grands arbres du “Branch and Cut” contiennent quelques centaines de nœuds, alors que dans l’algorithme du “Branch and Bound”, leur taille peut atteindre quelques millions

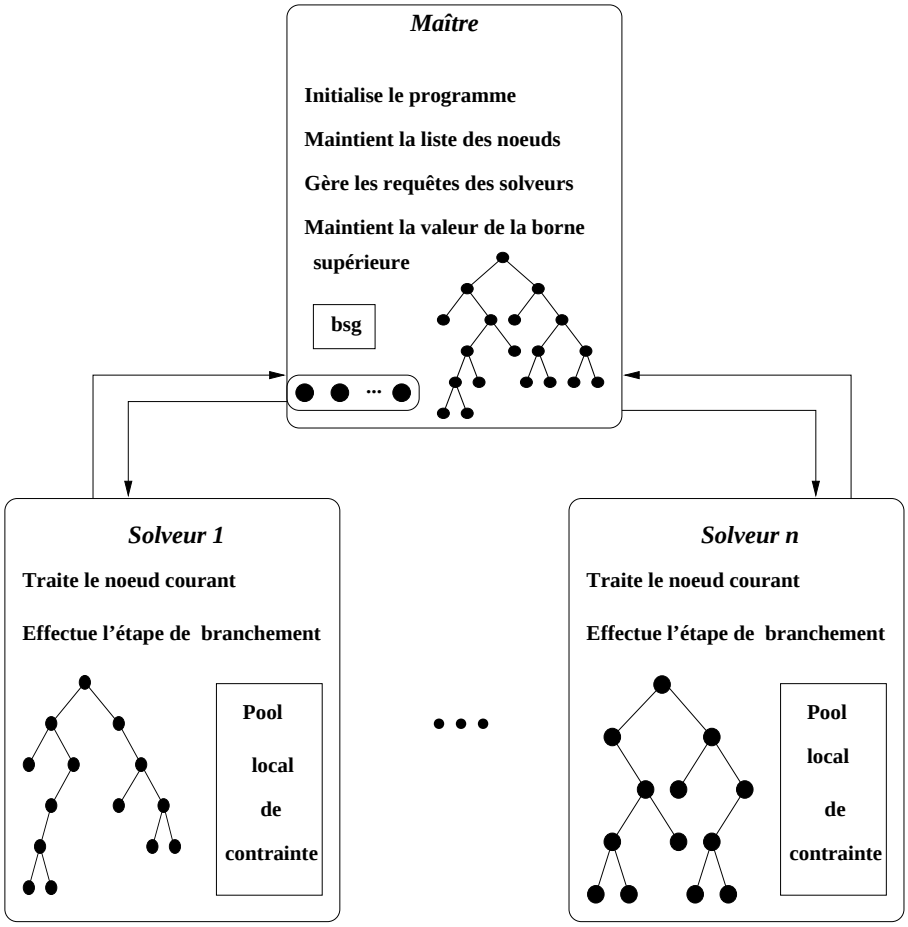

FIG. 6.2 – *Structure de l'algorithme parallèle du "Branch and Cut"*

de nœuds. Le nombre de processus utilisé lors de la parallélisation du "Branch and Cut" reste assez petit. Les risques de goulot d'étranglement autour d'un processus maître sont moindres à cause de l'espacement des requêtes des processus solveurs.

Dans notre implémentation, nous utilisons une architecture centralisée type maître-esclave qui gère la construction et le parcours de l'arbre du "Branch and Cut". La figure 6.2 montre la structure de notre algorithme. Nous définissons dans la suite les fonctionnalités des processus maître et esclaves.

Le processus maître

Le processus maître initialise le calcul, gère le parcours de l’arbre du “Branch and Cut” et le partage du travail entre les processus solveurs. Il maintient aussi la valeur de la borne supérieure. Et gère la terminaison de l’algorithme.

Les processus solveurs

Le processus solveur s’occupe de l’évaluation du nœud de l’arbre qui lui est affecté. Il sélectionne la contrainte ou la variable de branchement. Il génère des nouveaux nœuds fils, les évalue puis les communique au processus maître. Quand le solveur ne possède plus de nœud à traiter, il envoie une requête au maître pour lui demander du travail. L’évaluation d’un nœud de l’arbre est effectuée comme l’indique l’algorithme 4.

Dans notre algorithme séquentiel nous utilisons un pool de contraintes pour générer des contraintes violées par la solution courante du programme linéaire. Cette technique de séparation permet de générer facilement des contraintes sans passer par des heuristiques, très coûteuses en temps de calcul. Dans notre implémentation parallèle, nous utilisons des pools locaux de contraintes, au lieu d’un pool unique centralisé. En effet sachant que la génération de contraintes par le pool est assez fréquente, créer un pool centralisé induit des goulots d’étranglement inévitables autour du processus qui s’occupe de gestion du pool (en général, ce rôle est affecté au processus maître).

D’autre part, pour éviter la redondance de contraintes dans le programme linéaire qui induit sa dégénérescence, on vérifie si une contrainte voulant entrer dans le pool, n’y est pas déjà. Ainsi le travail du processeur gérant le pool centralisé augmente considérablement puisqu’il doit s’occuper de l’ajoût de nouvelles contraintes arrivant des processus esclaves, de la vérification de la non-redondance des contraintes et de la recherche de contraintes violées par des solutions courantes des processus esclaves. D’où l’intêret de l’utilisation de pools locaux. On a prévu dans notre implémentation la possibilité d’échanges de contraintes entre les pools locaux pour celles qui sont très difficiles à générer et dont le degré de violation par une solution fractionnaire est très grand.

Une autre structure possible du contrôle de la recherche arborescente consiste à utiliser une organisation hiérarchique de l’algorithme parallèle. Donc outre les

Algorithme 4 Evaluation d'un noeud de l'arbre du Branch and Cut

```
Continuer, nbcontr = 1;

Tant que (Continuer) faire
  Tant que (nbcontr et vallp < bsg et pas-d'essoufflement()) faire
    Tant que (nbcontr) faire
      Tant que (nbcontr) faire
        Résoudre le programme linéaire;
        Tant que (nbsoustour = rechercher-des-sous-tour()) faire
          Ajouter les contraintes de sous-tour au programme linéaire;
          Résoudre le programme linéaire;
        Fin Tant que /* boucle sous-tour */
        Si la solution est un tour alors
          Mettre à jour la valeur de la borne supérieure bsg;
          Exit;
        Fin Si
        nbcontr = rechercher-des-contraintes-violées-dans-le-pool();
      Fin Tant que /* boucle sous-tour + pool */
      Si moment-du-pricing alors
        nouvelles-variables = pricing();
        Si nouvelles-variables alors
          nbcontr = rechercher-des-contraintes-violées-dans-le-pool();
        Fin Si
      Fin Si
    Fin Tant que /* boucle sous-tour + pool + pricing */
    Elaguer-le-programme-linéaire();
    nbcontr = Rechercher-des-contraintes-par-les-heuristiques();
  Fin Tant que /* boucle recherche de contrainte */
  nouvelles-variables = pricing(); /* pricing avant branchement */
  Si nouvelles-variable= 0 alors
    Continuer = 0;
  Fin Si
Fin Tant que
```

processeurs maître et esclaves, on crée des processeurs auxiliaires pour chaque processeur esclave. Les processeurs auxiliaires s’occupent de la génération des contraintes par différentes méthodes ou de la recherche de la borne supérieure... Dans notre parallélisation, cela permet par exemple, de rechercher des contraintes violées par le pool et par les heuristiques de séparation en parallèle en utilisant des processeurs différents.

Malheureusement, dans ce type de structure hiérarchique, l’amélioration du temps de calcul se fait au détriment de l’utilisation effective des processeurs pendant la durée totale de l’exécution. Les processeurs passent une bonne partie de la durée de l’exécution à ne rien faire, puisque les tâches qui leur sont affectées ne consomment pas beaucoup de temps de calcul par rapport au temps global de l’exécution de l’algorithme (cf. le tableau 7.1). Ce type de stratégie induit des accélérations médiocres.

Un exemple d’implémentation avec une structure hiérarchique est décrit dans la thèse de Théodore K. Ralphs [Ral95]. C’est une parallélisation d’un code de “Branch and Cut” générique implémenté pour résoudre le problème de tournée de véhicules. La structure de l’algorithme du “Branch and Cut” parallèle est la suivante:

- *Master*: c’est le processeur maître, il joue un rôle annexe dans l’algorithme du “Branch and Cut”. Il gère les entrées/sorties de l’algorithme, possède toutes les données de l’instance résolue et les distribue selon les demandes des autres processeurs. Il calcule aussi la borne supérieure initiale, sauvegarde la meilleure solution et gère la terminaison de l’algorithme.

- *Tree Manager*: il maintient la liste des nœuds actifs. Il les distribue aux processeurs. Il maintient la valeur de la borne supérieure et informe les autres processeurs quand cette valeur change. Notons que ce processeur ne connaît ni le type du problème résolu, ni les données qui le concernent.

- *LP Solver*: c’est le processeur qui effectue le plus grand effort de calcul parmi tous les autres processeurs. Il résout les programmes linéaires du nœud courant de l’arbre du “Branch and Cut” et effectue l’opération de branchement. Il retourne les nouveaux nœuds au *Tree Manager*.

- *Cut Generator*: ce processeur ne s’occupe que de la génération de contraintes violées par la solution courante du programme linéaire par les heuristiques de séparations.

– *Cut Pool*: il maintient dans le pool une liste des contraintes valides. Il génère des contraintes violées par la solution courante du programme linéaire. Et réduit périodiquement la taille du pool.

Les dépendances entre chaque type de processeur sont indiquées dans la figure 6.4. L'auteur rapporte des exécutions sur IBM SP2 à 16 processeurs. Il utilise la librairie PVM [GBD+95].

6.3 Communication et mémoire

Dans un algorithme de "Branch and Cut" parallèle, la quantité et la taille de données échangées entre les processus sont importantes. Outre la valeur de la borne supérieure, un processus a besoin du programme linéaire de son nœud père et la contrainte de branchement pour pouvoir évaluer le nœud de l'arbre du "Branch and Cut". Notons que contrairement à l'algorithme du "Branch and Bound", les programmes linéaires résolus d'un nœud à l'autre diffèrent considérablement. D'une part par les contraintes générées au fur et à mesure de l'exploration de l'arbre et d'autre part par les variables du programme linéaire formée des arêtes du graphe partiel courant.

En effet, on commence l'exécution de l'algorithme avec un programme linéaire relaxé écrit sur un graphe partiel. On utilise ensuite la méthode de génération de colonnes pour ajouter de nouvelles variables dans le graphe partiel et la méthode des coupes polyédrales pour générer des contraintes violées. Quand on finit l'évaluation d'un nœud, on doit sauvegarder le programme linéaire. Ses lignes représentent l'ensemble des contraintes, et ses colonnes l'ensemble des variables. En général, les notions de contrainte et de ligne du programme linéaire sont équivalentes, de même pour colonne et variable.

Dans notre implémentation, une contrainte et une ligne sont des objets différents. Une contrainte est stockée indépendamment du graphe définissant les variables courantes. Un nœud possède un ensemble de contraintes actives qui appartiennent au programme linéaire courant. Quand une contrainte est ajoutée au programme linéaire alors on calcule sa ligne correspondante. Plus précisément une ligne est une représentation de la contrainte associée avec un certain ensemble de variables.

Pour bien comprendre cette différentiation prenons l'exemple des contraintes d'élimination de sous-tour. Ces contraintes sont définies sur un sous-ensemble S des sommets du graphe et s'écrivent sous la forme $x(\gamma(S)) \leq |S| - 1$. Si on

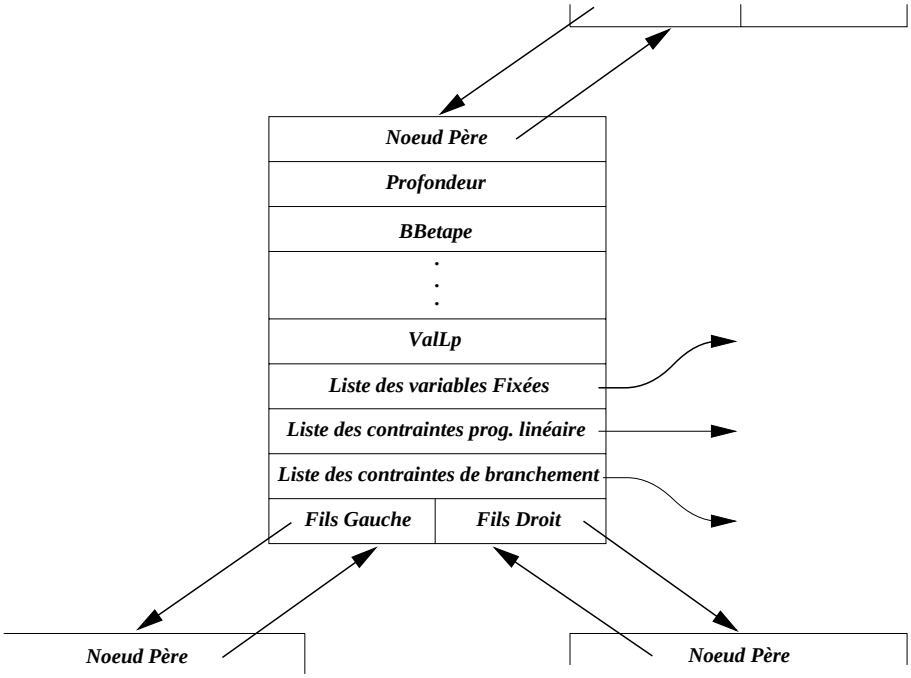

FIG. 6.3 – *Structure d’un noeud de l’arbre du “Branch and Cut”*

stocke ces inégalités de la même façon quand elles sont écrites dans le programme linéaire on devrait stocker toutes les arêtes (variables) dont les 2 extrémités sont dans S. Ce format utilise $O(|S|^2)$ place mémoire. Cependant il est aussi efficace de stocker les sommets de l’ensemble S ce qui nécessite $O(|S|)$ place mémoire. Etant donnée une arête e d’extrémités (u, v), ces arêtes valent 1 dans la contrainte d’élimination du sous tour si u et v appartiennent à S.

Cette différentiation permet de réduire le coût de communication et de stockage des contraintes. En effet au lieu de stocker tout le programme linéaire d’un nœud en attendant son traitement par un processus (ce qui impliquerait qu’on dispose d’un espace mémoire considérable pour stocker les programme linéaire de tous les nœuds actifs de l’arbre du “Branch and Cut”), on ne stocke que le squelette des contraintes définissant les lignes du programme linéaire et le graphe partiel sur lequel ce dernier est écrit. Notons que les contraintes sont stockées dans le pool sous leur forme réduite (squelettique).

Pour chaque nœud, on stocke l’ensemble des variables fixées pour la sous arborescence enracinée par le nœud. Ces variables seront communiquées au processus solveur lors de l’évaluation du nœud. La figure 6.3 montre la structure d’un

nœud de l'arbre du "Branch and Cut".

Notons qu'on ne stocke sur chaque nœud que les variables fixées, et les contraintes du programme linéaire qui sont générées pendant son évaluation. Pour reconstruire le programme linéaire d'un nœud quelconque, on doit partir du nœud racine de l'arbre et construire, au fur et mesure qu'on descend dans l'arbre vers le nœud courant, les contraintes du programme linéaire sur le graphe courant obtenu en appliquant toutes les fixations successives de chaque étape sur le graphe partiel de départ.

Quand le maître choisit un nœud actif pour être traité sur un processeur solveur, il n'est pas nécessaire de communiquer toute la branche. En effet chaque processeur possède l'arbre dont les feuilles forment l'ensemble des nœuds qui ont été résolu en local. Le maître ne communique que la sous branche qui manque au processeur solveur.

Le coût d'une communication se compose de deux parties, le temps de communication sur le réseau et le temps nécessaire à la reconstruction du programme linéaire d'un nœud. Ce coût est d'autant plus grand, quand on passe à un nœud très éloigné du nœud courant dans l'arbre du "Branch and Cut".

Notre implémentation prévoit aussi la reconstruction de la base du programme linéaire d'un nœud père pour ces fils, afin d'éviter de réappliquer la première phase du simplexe lors de la première résolution du programme linéaire dans un nœud. C'est un point majeur dans la structuration de notre implémentation parallèle. Il permet de réduire considérablement le temps de calcul.

6.4 Mécanismes d'équilibrage de charge

Dans un paradigme maître-esclave l'équilibrage de charge est équitable si on considère que les sous problèmes qui sont dans la file d'attente du maître sont indivisibles. Le processus maître donne du travail aux esclaves selon un ordre FIFO (*First In First Out*): Le premier processus qui envoie une requête demandant du travail au maître est le premier servi.

Cependant, au cours de l'exécution d'un "Branch and Cut" parallèle, plusieurs processus sont inactifs pendant le début de l'algorithme et pendant sa phase finale. Cette inactivité est due au nombre limité de sous problèmes qui peuvent s'exécuter d'une façon concurrente pendant un laps de temps plus ou moins important. La lenteur de l'évaluation d'un nœud dans la méthode "Branch and Cut" est la principale cause de ce phénomène.

Nous présentons dans ce paragraphe, les techniques que nous avons utilisées

pour augmenter le nombre de tâches concurrentes pendant les phases critiques du “Branch and Cut”.

Une manière de réduire le déséquilibre de charge est de diminuer le temps de l’évaluation d’un nœud et passer à la phase de branchement pour subdiviser le nœud en deux nouveaux sous problèmes. On crée ainsi deux nouvelles tâches concurrentes.

On arrête l’évaluation d’un nœud de l’arbre du “Branch and Cut” quand:

- La solution du programme linéaire est réalisable.
- La méthode de génération des coupes n’arrive plus à générer des contraintes violées.
- Il y a un essoufflement de la progression de la fonction économique.

Souvent, bien qu’on continue de générer des contraintes violées, la fonction économique à optimiser ne progresse plus. Ceci est principalement dû au fait qu’on ne génère plus à ce stade que des contraintes “voisines”: les hyperplans définis par les contraintes ont des normales quasiment parallèles, ainsi la solution du programme linéaire passe d’un sommet du polyèdre à un autre qui lui est voisin.

La détermination du degré d’essoufflement est une variable “subjective” définie par le programmeur. Dans une exécution séquentielle, on a l’habitude de pousser l’évaluation le plus loin possible avant de passer à la phase de branchement. Une manière pour équilibrer la charge de calcul est d’arrêter rapidement l’évaluation d’un nœud et le subdiviser en d’autres sous problèmes en choisissant judicieusement la fonction “essoufflement”.

La stratégie qu’on a adoptée consiste à changer la fonction essoufflement en fonction d’un certain seuil. Ce seuil dépend des nombres de sous problèmes en attente dans la file du manager et du nombre de solveurs utilisés. Lors d’une exécution parallèle, on utilise au début une fonction d’essoufflement “lâche” puisqu’il n’y a pas assez de sous problèmes à traiter, et dès qu’on arrive au seuil souhaité on change la fonction d’essoufflement pour pousser l’évaluation le plus loin possible.

Le choix du seuil et surtout de la fonction d’essoufflement doit être très minutieux puisqu’un mauvais choix peut augmenter considérablement le nombre de nœuds à traiter dans l’arbre du “Branch and Cut” et peut dégénérer l’algorithme en “Branch and Bound”, si la méthode de génération des coupes n’est pas assez utilisée.

Une autre méthode pour générer du calcul pour tous les processus est de jouer sur le nombre de sous problèmes générés lors du branchement. Cette méthode est aussi à utiliser avec précaution, puisqu'on a tendance à créer beaucoup de sous problèmes qu'on n'aurait pas traités pendant une exécution séquentielle avec une exploration du type "meilleur d'abord".

Dans des algorithmes d'énumération implicite, il est parfois préférable d'avoir de temps en temps des processeurs inactifs qu'un équilibrage de charge optimal avec une explosion du nombre de nœuds actifs de l'arbre de recherche.

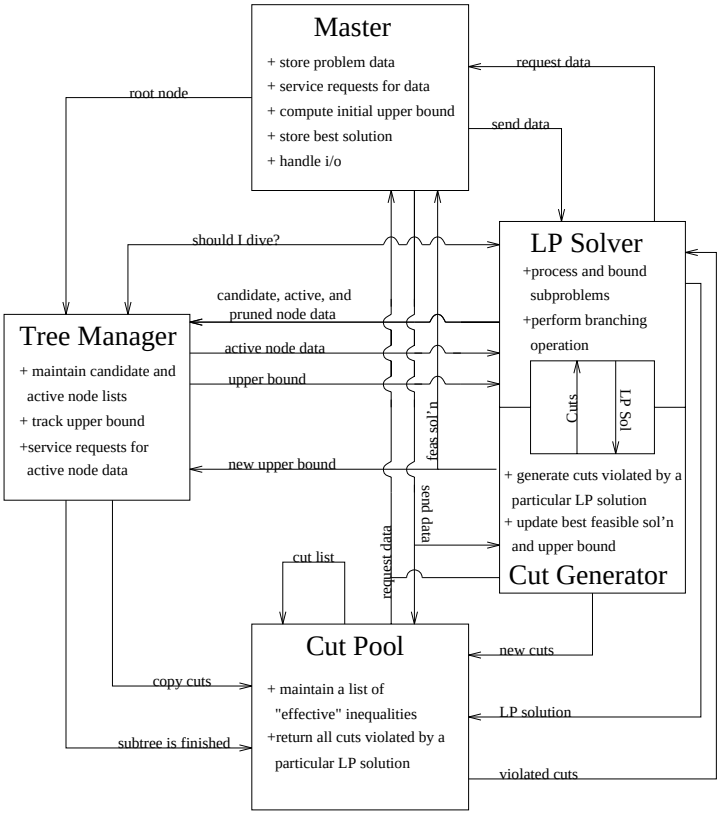

FIG. 6.4 – *Dépendances entre les types de processeurs dans la structure hiérarchique de T.K Ralphs*

Chapitre 7

Résultats et conclusions

Dans ce chapitre nous présenterons les résultats de nos expérimentations. Nous commencerons par justifier l'irrégularité du "Branch and Cut" en montrant son comportement séquentiel. Nous présenterons ensuite les résultats que nous avons obtenus avec notre algorithme parallèle. Nous expliquerons ces résultats en insistant sur les spécificités de notre implémentation.

7.1 Introduction

Toutes nos expérimentations ont été effectuées sur la machine parallèle IBM SP1 de l'IMAG. Notre implémentation parallèle de l'algorithme du "Branch and Cut" est écrite en C. Nous utilisons le logiciel CPLEX 4.0 [Cpl95] pour la résolution des programmes linéaires, et la bibliothèque standard de passage de message MPI [DA95] pour les communications.

MPI est une interface de programmation parallèle, par processus communicants. Elle possède des primitives de communication point-à-point et collectives, des primitives de gestion de groupe de processeurs, et de définition de topologie de communications.

Les instances du problème du voyageur de commerce résolues sont tirées de la librairie TSPLIB [Rei91]. Chaque instance porte un nom du type *XY* où *X* est un préfixe donnant l'origine de l'instance et *Y* représente le nombre de villes dans l'instance. Par exemple, l'instance $att48$ correspond à un problème de voyageur de commerce de 48 villes posé par la firme AT&T. Le nombre de variables dans cette instance est égal à $\frac{48\times47}{2} = 1128$.

Le IBM SP1 de l'IMAG est une machine à mémoire distribuée qui peut être

considérée comme un réseau de 32 stations RS/6000 avec 64 Mo de RAM. Les communications sont assurées par un réseau multi-étages du type “packet-switching” garantissant un débit full-duplex physique de 40 Mo/s pour chaque nœud et une latence inférieure à la microseconde. Les nœuds sont interconnectés par un switch rapide *High-Performance Switch* (HPS) TB2 qui possède des capacités DMA ainsi qu’un processeur (i860). Chaque nœud exécute une version du système UNIX AIX.

7.2 Comportement de l’algorithme du “Branch and Cut”

Dans ce paragraphe nous étudions le comportement de l’algorithme séquentiel du “Branch and Cut”. Cette étude tente de montrer la forte irrégularité de l’application, et donc de justifier certain de nos choix de parallélisation.

Dans la figure 7.1, on étudie l’évolution de la valeur de la solution du programme linéaire au cours du temps pour chaque nœud de l’arbre du “Branch and Cut” pour une instance de 439 villes. Chaque portion, ou morceau, de la courbe représente la variation de $vallp$ dans un nœud. Nous constatons que l’algorithme améliore considérablement la valeur de la borne inférieure au niveau de la racine de l’arbre du “Branch and Cut”. Cette valeur augmente lentement dans le reste de l’arbre. Nous remarquons aussi que la durée de l’évaluation de chaque nœud de l’arbre n’est pas prévisible et change constamment. Cette durée est représentée par la largeur de chaque morceau de la courbe. Elle a un comportement irrégulier et n’a pas forcément tendance à diminuer au fur et à mesure qu’on progresse dans l’arbre du “Branch and Cut”.

La figure 7.2 montre l’évolution du nombre de contraintes dans le programme linéaire au cours de l’exécution pour chaque nœud de l’arbre du “Branch and Cut”. Nous constatons que le nombre des contraintes augmente considérablement dans le nœud racine de l’arbre du “Branch and Cut”, ce qui explique l’amélioration de la valeur de la borne inférieure. On observe que le nombre des contraintes a tendance à augmenter au fur et à mesure qu’on progresse dans la résolution, sans exploser. Ceci est dû à l’élimination des contraintes non serrées du programme linéaire de temps en temps au cours de l’évaluation d’un nœud (l’effet est visible dans certaines étapes dans la figure). Dans l’exemple de la figure, on ne dépasse pas plus de 450 contraintes dans le programme linéaire en plus des 439 contraintes de degré initiales, malgré le nombre exponentiel de contraintes qui pourraient y

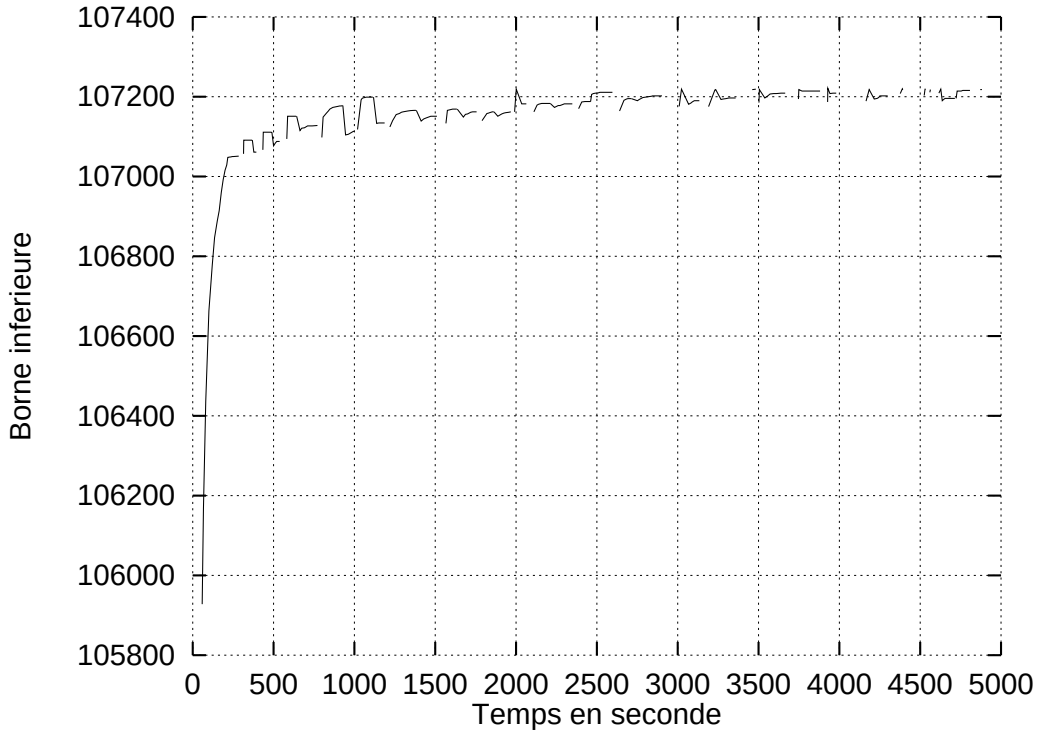

FIG. 7.1 – *Variation de la valeur $vallp$ au cours du temps dans chaque noeud de l'arbre du "Branch and Cut"*

entrer. Ceci nous rend optimiste sur la possibilité de résoudre de telles instances encore plus rapidement, si nous arrivions à construire des heuristiques plus performantes de séparation permettant de générer plus de contraintes violées.

La figure 7.3 montre l'évolution du nombre de variables dans le programme linéaire au cours de l'exécution pour chaque nœud de l'arbre du "Branch and Cut". Rappelons que dans notre code, nous initialisons le graphe partiel de départ par le graphe de triangulation de Daulaunay plus les arêtes de la meilleure solution réalisable connue. L'ajout d'une solution réalisable nous assure que le graphe de départ est hamiltonien, c'est à dire admet un cycle hamiltonien. On choisit d'ajouter la meilleure solution, en espérant qu'elle contient le maximum de variables de la solution optimale et donc de réduire le temps passé dans la génération de colonnes.

Nous remarquons dans la figure 7.3, que le nombre de variables a augmenté dans le nœud racine puis a diminué pour revenir très rapidement à un niveau proche de son niveau de départ. Ceci correspond à un ajustement de la qualité de l'ensemble des variables actives. L'algorithme a fait entrer plusieurs variables de coût réduit négatif pouvant appartenir à la solution optimale, ensuite, et au fur et à mesure qu'il progresse dans la résolution, et riche des informations obtenues lors de chaque évaluation, l'algorithme fixe plusieurs dizaines de variables à 0 ou 1 à chaque étape. Nous observons que pendant les dernières étapes de l'exécution, le programme linéaire ne possède qu'une centaine de variables en plus de 439 variables nécessaires à la construction d'une solution optimale, alors que le problème consistait à trouver un cycle hamiltonien avec 439 arêtes parmi 96141 arêtes au départ.

Les figures 7.2 et 7.3, nous confirment la forte irrégularité de la taille des données à communiquer pendant une exécution parallèle. Le nombre de variables et de contraintes dans le programme linéaire change considérablement d'un nœud à un autre et il est donc difficile de prévoir la taille des messages à passer à chaque processeur dans un algorithme parallèle. Rappelons ici que nous ne communiquons pas tout le programme linéaire dans notre implémentation parallèle, mais juste les contraintes et les variables ajoutées ou fixées dans chaque nœud.

Le tableau 7.1 donne une idée sur les pourcentages de la durée de l'exécution de chaque méthode composant l'algorithme du "Branch and Cut" par rapport à la durée totale de la résolution en séquentiel, pour trois instances de différentes tailles du problème du voyageur de commerce. La dernière ligne de ce tableau donne le pourcentage de la durée des résolutions du programme linéaire qui inclut entre autres, la résolutions des programmes linéaires pendant la recherche de la contrainte de branchement (notons que ces temps ont déjà été comptabi-

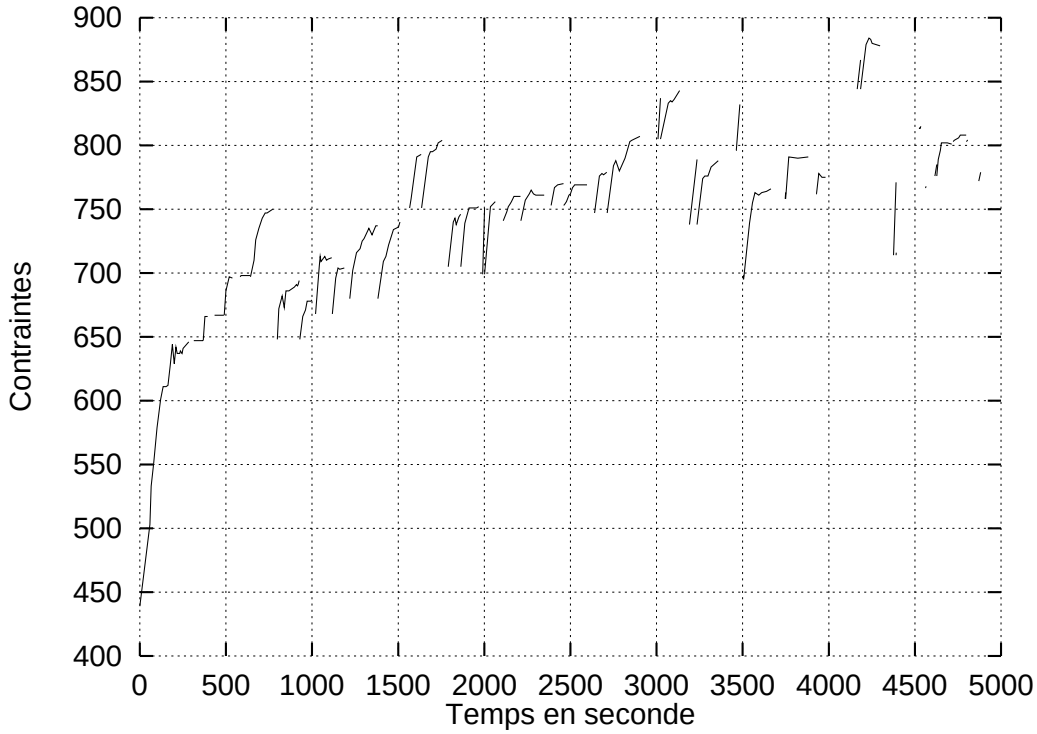

FIG. 7.2 – *Variation du nombre des contraintes dans chaque noeud de l'arbre du "Branch and Cut" au cours du temps*

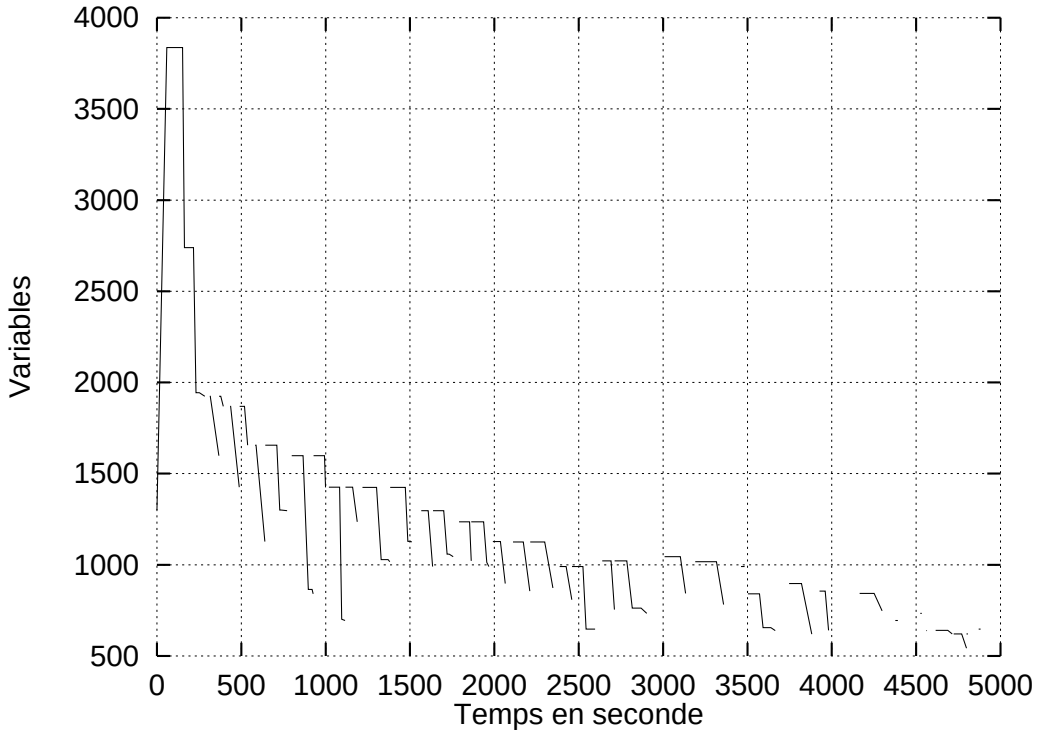

FIG. 7.3 – *Variation du nombre des variables dans chaque noeud de l'arbre du "Branch and Cut" au cours du temps*

lisé dans l'étape de branchement). Nous remarquons la prédominance de la phase {Résolution du programme linéaire + Séparation des sous-tours} qui représente dans les 3 cas au moins 40 % du temps global d'exécution. Ce qui confirme bien les raisons de l'inefficacité des structures hiérarchiques de l'algorithme parallèle du "Branch and Cut" présentées dans le chapitre précédent.

Etape de l'algorithme	pr76	att532	u724
Séparation des sous-tours	19.91	19.35	22.53
Heuristique de séparation	40.93	26.57	26.18
Séparation par le pool	5.54	12.83	21.86
Nettoyage du programme linéaire	0.21	0.08	0.06
Traitement des variables	9.37	7.70	8.40
Etape de branchement	10.01	22.31	13.90
Initialisation	0.70	0.04	0.03
Résolution du programme linéaire	21.71	35.96	24.50

TAB. 7.1 – *Pourcentage en temps de chaque étape de l'algorithme du "Branch and Cut" par rapport au temps total d'exécution*

7.3 Résultats expérimentaux

Avant de présenter des résultats numériques obtenus lors de la résolution du problème par la méthode du "Branch and Cut", nous allons donner quelques indices permettant d'intuiter ce qu'est une instance difficile du problème du voyageur de commerce.

Le nombre de villes dans une instances n'est sûrement pas la cause principale de la difficulté d'une instance. Plusieurs instances de taille raisonnable ne sont pas encore résolues alors que d'autres de plus grande taille le sont. La taille d'une instance implique un certain besoin de place mémoire pour résoudre les programmes linéaires, mais ne donne aucun a priori ni sur sur la taille de l'arbre du "Branch and Cut", ni sur le temps nécessaire à la résolution du problème. Ceci s'explique principalement par les différences de topologie des instances, c'est à dire la distribution des villes dans l'espace. Cette distribution influe sur la difficulté à générer des contraintes violées par l'algorithmes des coupes polyédrales, ou à obtenir de bonnes contraintes de branchement. Une topologie en grille comme pour le $ts225$ est très difficile à résoudre à cause de la symétrie des solutions optimales.

Des résultats expérimentaux de résolution du problème du voyageur de commerce par l'algorithme du "Branch and Cut" ont été publiés par Padberg et Rinaldi en 1991 [PR91], Clochard et Naddef en 1993 [CN93], Jünger Reinelt et Thienel en 1994 [JRT94] et Thienel en 1995 [Thi95]. Applegate, Bixby, Chvátal et Cook ont résolu des instances de très grande taille, cependant ces résultats ne sont pas encore publiés.

A titre indicatif nous donnons les résultats obtenus par Jünger, Reinelt et Rinaldi [JRR95] dans le tableau 7.2. Le temps CPU est donné en seconde sur une station SUN SPARC 10/20.

Instance	temps CPU	# nœuds
pr76	405	92
rd400	2511	54
pr439	3278	92
pcb442	530	50
rat575	7666	110
u724	9912	40

TAB. 7.2 – *Résultats expérimentaux obtenus par [JRR95]*

Le tableau 7.3 donne les résultats obtenus pour quelques instances difficiles de TSPLIB [Rei91]. Pour la majorité des instances résolues par notre code, l'arbre du "Branch and Cut" reste relativement petit grâce à l'éfficacité de nos heuristiques de sépération. Nous utilisons jusqu'à 8 processeurs esclaves pour résoudre des instances qui développent des arbres de grandes taille et jusqu'à 4 esclaves pour les autres. Nous constatons le bon comportement de l'algorithme. La perte d'efficacité observée pour quelques instances, quand on augmente le nombre de processeurs, est due principalement à l'inactivité de quelques esclaves pendant la phase initiale. Cette perte devient moins importante pour des instances qui génèrent plus de nœuds dans l'arbre du "Branch and Cut". Elle peut être causée, dans une moindre mesure, par l'augmentation de la taille de l'arbre du "Branch and Cut".

Nous avons essayé de résoudre l'instance $ts225$ qui est réputée très difficile sur 8 processeurs, malheureusement l'exécution s'arrête au bout d'une journée de calcul à cause du manque de mémoire sur les processeurs du SP1. L'exécution a généré un arbre du "Branch and Cut" de 1714 nœuds en donnant une solution avec une garantie de 0.0006% de l'optimale. Le taux d'occupation des processeurs dépasse en moyenne 99% et la répartition de la charge est quasiment parfaite.

Instance	# processeurs	Temps en s	# nœuds	Efficacité
pr76	1	226	16	1
	2	126	20	0.90
2850	3	83	14	0.91
variables	4	70	14	0.80
rd400	1	1341	14	1
	2	798	14	0.84
79800	3	621	14	0.72
variables	4	503	18	0.67
pr439	1	2415	30	1
	2	1462	34	0.83
96141	4	746	28	0.81
variables	8	432	38	0.70
pcb442	1	1166	28	1
	2	663	28	0.88
97461	4	352	22	0.83
variables	8	209	32	0.70
att532	1	2730	14	1
	2	1707	16	0.80
141246	3	1198	14	0.76
variables	4	990	16	0.69
rat575	1	5327	38	1
	2	3098	38	0.86
165025	4	1645	40	0.81
variables	8	925	46	0.72
u724	1	6056	26	1
	2	3403	28	0.89
261726	4	2046	30	0.74
variables	8	1306	42	0.58

TAB. 7.3 – *Variation du temps de calcul, de l'efficacité et du nombre de noeuds dans l'arbre du "Branch and Cut" selon le nombre de processeurs utilisés pour résoudre quelques instances difficiles du problème du voyageur de commerce*

Nous observons aussi que le surcoût dû à la communication des données provoqué par la parallélisation n'est pas important. Ceci est dû à l'optimisation des communications lors du passage des données du maître à un nœud esclave. Notons que le coût des communications n'inclut pas uniquement le coût de la communication des données sur le réseau. En effet, il n'existe pas de primitives directes pour les communications des listes dans la librairie de passage de messages MPI. Nous utilisons alors les primitives MPI-pack et MPI-unpack qui induisent un surcoût non négligeable. C'est en additionnant les coûts de transmission des données dans le réseau et le coût des phases de paquetage et dépaquetage des données, qu'on obtient le coût réel des communications.

La communication des données et le dépaquetage sur le processeur cible sont effectués de façon à permettre la reconstruction de la base du programme linéaire du nœud père. Ceci permet d'économiser le temps de calcul de la première phase du simplexe lors de la résolution du programme linéaire. D'où un gain substantiel de temps, vu le pourcentage qu'occupe la résolution du programme linéaire dans le temps total d'exécution de l'algorithme du "Branch and Cut".

L'étude de la trace de notre programme parallèle avec 2 processeurs nous montre que les esclaves restent sans activité pendant la phase d'initialisation du maître puisqu'ils sont présents dans l'environnement parallèle dès le début de l'exécution. En effet, contrairement à PVM, MPI ne permet pas de faire entrer des nouveaux processeurs pendant l'exécution. La version suivante de la librairie standard de passage de message MPI2, devrait permettre de le faire.

La quatrième colonne du tableau 7.3 donne la taille de l'arbre du "Branch and Cut" pour chaque exécution. Nous constatons que, pour une instance donnée, la taille de l'arbre est différente selon le nombre des processeurs que nous utilisons, ceci est dû principalement à l'utilisation de pools locaux de contraintes. En effet, la construction du pool locale est très dépendante des programmes linéaires résolus sur le processeur. Or selon le nombre de processeurs utilisés les nœuds traités sur un processeur sont différents, donc il arrive qu'on génère une contrainte du pool pendant une exécution qu'on n'aurait pas générée dans une autre exécution, et par chance cette contrainte améliore rapidement la borne inférieure d'un nœud et donc fait diminuer la taille de l'arbre.

Notons au passage que nous utilisons les heuristiques de recherche de contraintes violées d'une manière hiérarchique. On cherche d'abord les contraintes de type peigne, si aucune contrainte de peigne n'est violée, on cherche les autres types de contraintes. Notre code parallèle n'utilise que les heuristiques pour la génération de contrainte de peigne et de chemin pour ne pas réduire considérablement la taille de l'arbre des instances que nous résolvons sur le SP1.

L'utilisation de branchement sur les contraintes permet de trouver des branchements forts et donc permet de ne pas faire exploser la taille de l'arbre comme dans le "Branch and Bound". De même, l'utilisation d'une heuristique performante d'exploitation de la solution fractionnaire permet de trouver rapidement une bonne borne supérieure, et donc d'arrêter rapidement l'exploration d'une branche de l'arbre du "Branch and Cut". Notre heuristique [Bou95] exploite les solutions fractionnaires du programme linéaire pour construire un cycle de bonne qualité, ensuite l'améliore avec un algorithme de Lin-Kernigham modifié. Pour plusieurs instances, on obtient la valeur de la solution optimale dès la racine de l'arbre du "Branch and Cut".

Notons que les anomalies d'accélération observées pour les algorithmes du "Branch and Bound" parallèles sont très rares dans le cas du "Branch and Cut", malgré l'exploration selon une stratégie meilleure d'abord. Ceci est dû principalement à l'utilisation d'un algorithme d'exploitation d'une solution factionnaire qui donne très rapidement une très bonne borne supérieure au cours de l'exécution.

Notons enfin que nous avons réussi à résoudre plusieurs instances du problème du voyageur de commerce directement au nœud racine de l'arbre du "Branch and Cut". D'autres instances réputées difficiles ont été résolues en quelques dizaines de nœuds. Et celles de très grande taille n'ont pas pu être résolues puisqu'elles utilisent beaucoup de mémoire lors de la résolution du programme linéaire, et donc atteignent très vite la limite de la mémoire des processeurs du SP1.

Equilibrage de charge

Le tableau 7.4 donne les résultats obtenus en exécutant l'algorithme du "Branch and Cut" sans aucun mécanisme d'équilibrage de la charge entre les processeurs, et avec mécanismes d'équilibrage de la charge en utilisant une stratégie de seuil sur la fonction essoufflement ou une stratégie de multi-branchements. Nous remarquons que les deux stratégies induisent un surcoût dans la recherche arborescente. La stratégie de seuil sur la fonction d'essoufflement induit un surcoût plus faible que celui de la stratégie de multi-branchement et arrive à une meilleure répartition de la charge du travail entre les processeurs. La stratégie de multi-branchement peut faire exploser la taille de l'arbre du "Branch and Cut", puisqu'elle a tendance à créer des sous-problèmes qui ne sont pas visités dans une exécution séquentielle.

Notons en outre que le choix du seuil dans la stratégie d'essoufflement est un point très important pour le bon déroulement de l'algorithme. En effet, nous avons constaté dans la figure 7.1 que la valeur de la borne inférieure augmente lentement pendant l'évaluation d'un nœud dès que nous descendons un peu dans

Mode d'équilibrage	Instance	Temps en s	# nœuds	Efficacité
Aucun	pr439	451	32	0.67
	pcb442	209	32	0.70
	rat532	1057	40	0.63
	u724	1514	26	0.50
Seuil sur la fonction d'essoufflement	pr439	432	38	0.70
	pcb442	209	32	0.70
	rat532	925	46	0.72
	u724	1306	42	0.58
multi-branchement	pr439	444	38	0.68
	pcb442	218	42	0.67
	rat532	965	50	0.69
	u724	1514	50	0.50

TAB. 7.4 – *Comparaison des résultats obtenus en exécutant l'algorithme du "Branch and Cut" sur 8 processeurs avec des modes d'équilibrage de charge différents*

l'arbre. Ainsi si on utilise une fonction d'essoufflement "lâche" ailleurs qu'au début de l'exécution, alors on passera à chaque fois très rapidement à la phase de branchement, et on perdra la spécificité de l'algorithme du "Branch and Cut", qui consiste à privilégier la recherche des contraintes violées sur l'énumération des solutions comme dans le "Branch and Bound". Dans ce cas on observe une explosion de la taille de l'arbre à visiter.

Notons néanmoins que pour des instances qui construisent des grands arbres de "Branch and Cut", comme pour le $ts225$, l'exécution de l'algorithme sans mécanismes d'équilibrage de charge donne une bonne répartition du calcul entre les processeurs et des efficacités proche de 1. L'utilisation des algorithmes d'équilibrage n'est nécessaire que pour des instances qui génèrent des petits arbres de recherche.

Conclusions et perspectives

Dans cette thèse nous nous sommes intéressés à l'étude et la parallélisation de l'algorithme du "Branch and Cut" afin de résoudre jusqu'à l'optimalité le problème du voyageur de commerce.

Dans la première partie de ce travail, nous avons présenté les composantes principales de l'algorithme du "Branch and Cut". Nous avons étudié ensuite le problème du voyageur de commerce par une approche polyédrale. Nous avons donné une description détaillée de notre implémentation de l'algorithme du "Branch and Cut" pour résoudre ce problème jusqu'à l'optimalité.

Ces thèmes ont été abordés dans les trois premiers chapitres de cette thèse. Nous avons présenté en particulier dans le troisième chapitre, les phases d'initialisation et de résolution du programme linéaire. Nous avons décrit ensuite, la génération de coupes en utilisant des heuristiques de séparation et un pool de contraintes. Nous avons introduit une heuristique de recherche de borne supérieure au problème du voyageur de commerce. Puis nous avons présenté l'étape de génération de colonnes et les possibilités de fixations de variables. Nous avons terminé par une description de l'étape de branchement.

Dans la deuxième partie de cette thèse, nous avons résumé les principes de base du parallélisme, et nous avons présenté un état de l'art des études menées sur la parallélisation de l'algorithme du "Branch and Bound". Nous avons décrit ensuite plusieurs modèles de parallélisation de l'algorithme du "Branch and Cut" selon le niveau de granularité utilisé.

Fort des connaissances acquises, nous avons développé un programme de "Branch and Cut" parallèle pour résoudre le problème du voyageur de commerce. Nous avons adopté une stratégie centralisée de contrôle de la recherche arborescente. Nous avons mis en place par un choix judicieux de la représentation des données, un mécanisme de minimisation des coûts liés aux différentes étapes de la communication entre les processeurs. Nous avons également mis en place plusieurs stratégies d'ajustement de la charge au début de l'exécution du "Branch and

Cut", et avons montré l'intérêt de pouvoir adapter la granularité en fonction des paramètres du programme (fonction d'essoufflement...) et du nombre des processeurs disponibles.

En résumé, nous avons réussi grâce au parallélisme, à réduire le temps de résolution de plusieurs instances difficiles du problème du voyageur de commerce. Nous avons aussi contribué, à notre connaissance, aux premières tentatives de parallélisation de l'algorithme du "Branch and Cut".

Ainsi, pour résoudre de grandes instances de problèmes $\mathcal{NP}$-difficiles, il s'avère intéressant de poursuivre l'étude de la parallélisation de cette méthode. Nous comptons continuer nos recherches sur deux axes:

D'une part en travaillant sur l'amélioration de notre code parallèle. En effet, il serait intéressant de passer vers un grain de parallélisme plus fin pour pouvoir accélerer la résolution des instances qui se résolvent sans passer à la phase de branchement. Il est aussi intéressant d'utiliser la multiprogrammation légère pour pouvoir entrelacer les calculs et les communications.

Et d'autre part, en améliorant les différentes étapes de l'algorithme du "Branch and Cut". Nous considérons notamment que la recherche de la contrainte de branchement est l'une des étapes de l'algorithme qui peut encore être considérablement améliorée.

Bibliographie

[AGF+95] Autie (Gerard), Garcia (Jean-Maurice), Ferreire (Afonso), Roch (Jean-Louis), Villard (Gilles), Roman (Jean), Roucairol (Catherine) et Virot (Bernard). – *Parallélisme et applications irrégulières*. – Hermes, 1995.

[BC91] Boyd (S.C.) et Cunnigham (W.H.). – Small traveling salesman polytopes. *Mathematics of Operations Research*, vol. 16, 1991.

[BCC93a] Balas (Egon), Ceria (Sebastian) et Cornuéjols (Gerard). – A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, vol. 58, 1993, pp. 295–324.

[BCC93b] Balas (Egon), Ceria (Sebastian) et Cornuéjols (Gerard). – Solving mixed 0-1 programs by a lift-and-project method. *In : Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 232–242.

[BCC94] Balas (Egon), Ceria (Sebastian) et Cornuéjols (Gerard). – *Mixed 0-1 programming by lift-and-project in a branch-and-cut framework*. – Rapport technique, Management Science Research Report MSRR-603, GSIA, Carnegie Mellon University, Pittsburgh, 1994.

[BCQW93] Boyd (S.C.), Cunnigham (W.H.), Queyranne (M.) et Wang (Y.). – Ladders for the traveling salesman. *SIAM Journal on Optimization*, vol. 5, 1993.

[BKR94] Burkhard (Rainer), Karisch (Stefan) et Rendl (Franz). – *QAPLIB—A quadratic assignment problem library*. – Rapport technique n° 287, Technische Universität Graz, Institut für Mathematik, 1994.

[BMCP97] Brüngger (A.), Marzetta (A.), Clausen (J.) et Perregaard (M.). – Joining forces in solving large-scale quadratuc assignment problems in parllel. *In : Proceedings of the 11th IPPS*.

[BMCP98] Brüngger (A.), Marzetta (A.), Clausen (J.) et Perregaard (M.). – Solving large-scale qap problems in parallel with the search library zram. *Journal of Parallel and Distributed Computing*, 1998, pp. 157–169.

[BMFN96] Brüngger (A.), Marzetta (A.), Fukuda (K.) et Nievergelt (J.). – The parallel search bench zram and its applications. – December 1996. CroSCutS.

[Bou95] Bouzgarrou (M. E.). – *Exploitation d'une solution fractionnaire d'un algorithme du "Branch and Cut"*. – These de dea, Laboratoire ARTEMIS-IMAG, Universit Joseph Fourrier de Grenoble, 1995.

[CF95a] Corrêra (R.) et Ferreira (A.). – On the effectivness of synchronous branch-and-bound algorithms. *Parallel Processing Letters*, vol. 5, n° 3, 1995, pp. 375–386.

[CF95b] Corrêra (Ricardo) et Ferreira (Afonso). – A distributed implementation of asynchronous parallel branch-and-bound. *In : Parallel Algorithms for Irregular Problems: State of the Art*, éd. par Ferreira (A.) et Rolim (J.). – Kluwer Academic publishers, 1995.

[CFN85] Cornuéjols (Gérard), Fonlupt (Jean) et Naddef (Denis). – The traveling salesman problem on a graph and some related polyhedra. *Mathematical Programming*, vol. 33, 1985, pp. 1–27.

[Chv73] Chvátal (Vasek). – Edmonds polytopes and weakly hamiltonian graphs. *Mathematical Programming*, vol. 5, 1973, pp. 29–40.

[Chv83] Chvátal (Vasek). – *Linear Programming*. – W.H. Freeman and Company, 1983.

[CN93] Clochard (Jean Maurice) et Naddef (Denis). – Using path inequalities in a branch-and-cut code for the symmetric traveling salesman problem. *In : Proceedings on the Third IPCO Conference*, éd. par Wolsey (Lawrence) et Rinaldi (Giovanni), pp. 291–311.

[Cpl95] Cplex. – *Using the Cplex Callable Library*. – Cplex Optimization, Inc, 1995.

[CR95] Christof (T.) et Reinelt (G.). – Parallel cutting plane generation for the tsp. – 1995. Extended Abstract.

[CT93] Cosnard (Michel) et Trystram (Denis). – *Algorithmes et architectures paralleles*. – InterEditions, 1993.

[DA95] Dongarra et Al. – *MPI: A Message-Passing Interface Standard*. – University of Tennessee, Knoxville, Tenessee, June 1995. available via www.netlib.org/mpi/.

[DFJ54] Dantzig (George B.), Fulkerson (D. Ray) et Johnson (Selmer M.). – Solution of a large scale traveling salesman problem. *Operations Research*, vol. 2, 1954, pp. 393–410.

[DLCMM96] Denneulin (Yves), Le Cun (Bertrand), Mautor (Thierry) et Méhaut (Jean-Francois). – Distributed branch and bound algorithms for large quadratique assignement problems. *In : 5th Computer Science Technical Section on Computer Science and Operations Research*.

[Duc90] Ducan (R.). – A survey of parallel computer architectures. *IEEE Computer*, vol. 23(2), 1990, pp. 5–16.

[dV97] de Vitis (Andrea). – *The cactus representation of all minimum cuts in a weighted graph*. – Rapport technique, Viale Manzoni 30, 00185 Roma, Italy, Insituto di Analisi dei Sistemi ed Informatica del CNR, 1997.

[Fle88] Fleischmann (B.). – A new class of cutting planes of the symmetric traveling salesman problem. *Mathematical Programming*, vol. 40, 1988, pp. 225–246.

[Fle97] Fleischer (Lisa). – *Building chain and cactus representations of all minimum cuts from Hao-Orlin in the same asymptotic run time*. – Rapport technique, Cornell University, School of Operations Research and Industrial Engineering, 1997.

[Fly79] Flynn (M.J.). – Some computer organisation and their effectiveness. *IEEE Transaction on computer*, 1979, pp. 948–960.

[GBD+95] Geist (A.), Beguelin (A.), Dongarra (J.), Jiang (W.), Mancheck (R.) et Sunderam (V.). – *PVM3 User's Guide and Reference Manual*. – Oak Ridge Natinal Laboratory, Mai 1995.

[GC94] Gendron (Bernard) et Crainic (Teodor Gabriel). – Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, vol. 42, n° 6, November-December 1994, pp. 1042–1066.

[GJR84] Grötchel (M.), Jünger (M.) et Reinelt (G.). – A cutting plane algorithm for the linear ordering problem. *Operations Research*, no32, 1984, pp. 1195–1220.

[GLS88] Grötschel (M.), Lovász (L.) et Schrijver (A.). – *Geometric Algorithms and Combinatorial Optimization*. – Springe-Verlag, Berlin-Heidelberg, 1988.

[Gom58] Gomory (Ralph E.). – Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, vol. 64, 1958, pp. 275–278.

[GP79] Grötschel (Martin) et Padberg (Manfred W.). – On the symmetric traveling salesman problem II: Lifting theorems and facets. *Mathematical Programming*, vol. 16, 1979, pp. 281–302.

[GP86] Grötschel (Martin) et Pulleyblank (William). – Clique tree inequalities and the symmetric traveling salesman problem. *Mathematics of Operations Research*, vol. 11, 1986, pp. 537–569.

[IBM95] IBM. – *Optimization Subroutine Library - Guide and Reference, Release 2.1*. – IBM Corporation, 1995.

[JAM88] Janakiram (V.K.), Agrawal (D.P.) et Mehrotra (R.). – A randomized parallel branch-and-bound algorithm. *In : The 1988 International Conference on Parallel Processing*, pp. 69–75.

[JRR95] Jünger (Michael), Reinelt (Gerhard) et Rinaldi (Giovanni). – The traveling salesman problem. *In : Network Models*, éd. par Ball (M.), Magnanti (T.), Monma (C.L.) et Nemhauser (G.L.), pp. 225–330. – Amsterdam, North Holland, 1995.

[JRT94] Jünger (Michael), Reinelt (Gerhard) et Thienel (Stefan). – Provably good solutions for the traveling salesman problem. *Zeitschrift für Operations Research*, vol. 40, 1994, pp. 183–217.

[KK84] Kumar (V.) et Kanal (L.N.). – Parallel branch-and-bound formulations for and/or tree search. *IEEE Trans.Pattern Anal. and Mach. Intel.*, 1984, pp. 768–778.

[KP82] Karp (Robert M.) et Papadimitriou (Christos H.). – On linear characterizations of combinatorial optimization problems. *SIAM Journal on Computing*, vol. 11, 1982, pp. 620–632.

[KRNR88] Kumar (Vipin), Ramesh (K.) et Nageshwara Rao (V.). – Parallel best-first search of state-space graphs: Asummury of results. *In : 7th National Conference on Artificial Intelligence*, pp. 122–127.

[LCR95] Le Cun (B.) et Roucairol (C.). – *BOB: a Unified Platform for Implementing Branch-and-Bound Algorithms*. – Rapport de Recherche n° 95/16, Laboratoire PRISM, Universite de Versailles Saint-Quentin-en-Yvelines, 1995.

[LMS94] Lustig (I.), Marsten (R.E.) et Shanno (D.F.). – Interior point methods for linear programming: computational state of the art. *ORSA Journal on computing*, vol. 6, 1994, pp. 1–14.

[LS84] Lai (T.-H.) et Sahni (S.). – Anomalies in paralllel branch-and-bound algorithms. *Communication ACM*, vol. 27, June 1984, pp. 594–602.

[LS85] Lai (T.-H.) et Sprague (A.). – Performance in parllel branch-and-bound algorithms. *IEEE Trans. Comput.*, vol. C-34, 1985, pp. 962–964.

[MP89] Miller (D.L.) et Pekny (J.F.). – Results from a parallel branch and bound algorithm for the asymmetric traveling salesman problem. *Operations Research Letters*, no8, 1989, pp. 129–135.

[MP93] Miller (D.L.) et Pekny (FJ.F.). – The role of performance metrics for parallel mathematical programming algorithms. *ORSA J. Comput*, vol. 5, 1993, pp. 26–28.

[MR96] Mans (B.) et Roucairol (C.). – Performances of parallel branch and bound algorithms with best-first search. *Discrete Applied Mathematics*, vol. 66, 1996, pp. 57–76.

[Nad92] Naddef (Denis). – The binested inequalities of the symmetric traveling salesman polytope. *Mathematics of Operations Research*, vol. 17, 1992, pp. 882–900.

[NI92] Nagamochi (H.) et Ibaraki (T.). – Computing edge-connectivity in multi-graphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, vol. 5, 1992, pp. 54–66.

[NP98] Naddef (Denis) et Pochet (Yves). – *The traveling salesman polytope revisited*. – Rapport technique, Université Joseph Fourier, Grenoble, 1998. to appear.

[NR91] Naddef (Denis) et Rinaldi (Giovanni). – The symmetric traveling salesman polytope and its graphical relaxation: Composition of valid inequalities. *Mathematical Programming*, vol. 51, 1991, pp. 359–400.

[NR92] Naddef (Denis) et Rinaldi (Giovanni). – The crown inequalities for the traveling salesman polytope. *Mathematics of Operations Reserach*, vol. 17, 1992, pp. 308–326.

[NR93] Naddef (Denis) et Rinaldi (Giovanni). – The graphical relaxation: a new framework for the symmetric traveling salesman polytope. *Mathematical Programming*, vol. 58, 1993, pp. 53–88.

[NT98a] Naddef (Denis) et Thienel (Stefan). – *Efficient separation routines for the symmetric traveling salesman problem I: general tools and comb separation*. – Rapport technique, Universität zu Köln, 1998. to appear.

[NT98b] Naddef (Denis) et Thienel (Stefan). – *Efficient separation routines for the symmetric traveling salesman problem II: separating multi handle inequalities*. – Rapport technique, Universität zu Köln, 1998. to appear.

[PM92] Pekny (J.F.) et Miller (D. L.). – A parallel branch-and-bound algorithm for solving large asymetric traveling salesman problems. *Mathematical Programming*, vol. 55, 1992, pp. 17–33.

[PR82] Padberg (Manfred W.) et Rao (M. Ram). – Odd minimum cut sets and b-matchings. *Mathematics of Operations Research*, vol. 7, 1982, pp. 67–80.

[PR87] Padberg (Manfred W.) et Rinaldi (Giovanni). – Optimization of a 532 city symmetric traveling salesman problem by branch-and-cut. *Operations Research Letters*, 1987, pp. 1–7.

[PR90] Padberg (Manfred W.) et Rinaldi (Giovanni). – An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, vol. 47, 1990, pp. 19–36.

[PR91] Padberg (Manfred W.) et Rinaldi (Giovanni). – A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, vol. 33, 1991, pp. 60–100.

[QD85] Quinn (M.J.) et Deo (N.). – An upper bound for the speedup of parallel branch and bound algorithms. *In : Proceedings of the 3rd Conf. on Found. of Software Technology and Theorical Computer Science*, pp. 488–504. – Bangalore, India, 1985.

[Qui90] Quinn (Michael J.). – Analysis and implementation of branch and bound algorithms on a hypercube multicomputer. *IEEE Transactions on Computers*, vol. 39, n° 3, mars 1990, pp. 384–387.

[Ral95] Ralphs (T.K.). – *Parallel Branch and Cut for vehicule routing*. – Thèse de PhD, Cornell University, 1995.

[Rei91] Reinelt (Gerhard). – TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing*, vol. 3, 1991, pp. 376–384.

[Rou87] Roucairol (Catherine). – *Du Séquentiel au paralèlle: la recherche arborescente et son application à la programmation quadratique en varibales 0-1*. – Thèse d'État, Université Paris VI, France, 1987.

[Rou96] Roucairol (Catherine). – Parallel computing for difficult combinatorial optimization problems. *European Journal of Operation Research*, vol. 92, 1996, pp. 573–590.

[Sch86] Schrijver (A). – *Theory of linear and integer programming*. – John Wiley & Sons, 1986.

[SHH97] Shinano (Yuji), Harada (Kenichi) et Hirabayashi (Ryuichi). – Control schemes in a generalized utility for parallel branch-and-bound algorithms. *In : 11th International Parallel Processing Symposium*. pp. 621–627. – IEEE Computer Society Press.

[Thi95] Thienel (Stefan). – *ABACUS—A Branch-And-CUt System*. – Thèse de PhD, Universität zu Köln, 1995.

[TLM95] Tschöke (S.), Lüling (R.) et Monien (B.). – Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 12024 processor network. *In : 9th international Parallel Processing Synposium (IPPS'95)*. pp. 1832–189. – IEEE Computer Society Press.

Résumé :

La résolution jusqu'à l'optimalité de problèmes d'optimisation combinatoire $\mathcal{NP}$-difficiles nécessite une mise en œuvre de méthodes de plus en plus complexes qui consomment de plus en plus de puissance de calcul. L'objectif de notre travail est de paralléliser un algorithme de "Branch and Cut" pour résoudre jusqu'à l'optimalité des instances difficiles du problème du voyageur de commerce.

Dans la première partie de notre travail, nous présentons les composantes principales de l'algorithme du "Branch and Cut". Nous étudions ensuite le problème du voyageur de commerce par une approche polyédrale. Nous donnons enfin une description détaillée de notre implémentation de l'algorithme du "Branch and Cut".

Dans la deuxième partie, Nous commençons par une brève présentation du parallélisme, et un état de l'art des études menées sur la parallélisation de l'algorithme du "Branch and Bound". Puis, nous proposons plusieurs modèles de parallélisations de l'algorithme du "Branch and Cut". Nous décrivons ensuite la stratégie de contrôle de la recherche arborescente qu'on a adopté, les mécanismes de minimisation des coûts liés aux différentes étapes de la communication entre les processeurs et les stratégies d'équilibrages. Nous terminons en donnant les résultats obtenus sur le IBM-SP1.

Mots clés : parallélisme, "Branch and Cut and Price", problème du voyaguer de commerce.

Abstract:

The Branch and Cut method has been very successful in solving large Symmetric Traveling Salesman instances to optimality. The time needed to solve these instances is very high, therefore one may want to consider the possibility of using parallel machines to perform the computations.

In the first part of this thesis, we present different components of Branch and Cut. We give a polyhedra study of the STSP. We then describe our implementation of the Branch and Cut method.

In the second part of this document, we start with a presentation of some basic concepts of parallelism and the state of the art of the parallelization of the branch-and-bound algorithm which is a related method. Next we propose several models to parallelize the Branch and Cut. We present the choices we have made when we have implemented our parallel algorithm: control scheme, load balancing strategy... Finally, we give many computational results on a IBM-SP1 machine.

Keywords: parallelism, Branch and Cut and Price, Symmetric Traveling Salesman Problem