RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ HASSIBA BENBOUALI-CHLEF (U.H.B.C)

FACULTÉ DES SCIENCES EXACTES ET D'INFORMATIQUE

Domaine : Mathématique et Informatique

Filière : Mathématique

Spécialité : Mathématique Appliquées et Statistique

MÉMOIRE

pour l'obtention du diplome de

MASTER

ALGORITHME BRANCH AND BOUND ET APPLICATION AU PROBLÈME TSP

Présenté par :

AHMEDI EZZOURGUI ZAHIA

KALLOUCH SALMA

Soutenu publiquement le : 02/06/2016, devant le jury composé de :

Année Universitaire :2015/2016

Remerciement

Tout d'abord, nous remercions le Dieu, notre créateur de nos avoir donné les forces, la volonté et le courage an d'accomplir ce travail modeste.

Nous adressons le grand remerciement notre encadreur **Mr. BELGACEM Rachid** qui a proposé le thème de ce mémoire, pour ses conseils et ses dirigés du début à la fin de ce travail.

Nous tenons également remercier messieurs les membres de jury pour l'honneur qu'ils nos ont fait en acceptant de siéger notre soutenance, tout particulièrement :

Mem. FZ.Belkerdim pour nous avoir fait l'honneur de présider le jury de cette mémoire. Nous souhaitons exprimer notre gratitude **Mr.A. Mezouaghi** pour avoir faire de lecteur notre mémoire, aller l'examiner et il peut évaluer cette mémoire. Nous vous remercions pour l'intérêt que vous avez porté ce travail et pour vos précieux conseils et remarques.

Finalement, nous tenons exprimer notre profonde gratitude nos familles qui nous ont toujours soutenues et tout ce qui participe de réaliser ce mémoire. Ainsi que l'ensemble des enseignants qui ont contribué notre formation.

Table des matières

Introduction

Quelque soit son domaine, l'être humain est confronté à différents problèmes dans toutes les sphères de la société. Un problème donné peut être défini par l'ensemble des propriétés que doivent vérifier ses solutions. Il peut être un problème de décision ou un problème d'optimisation.

En pratique, il arrive fréquemment que dans un problème d'optimisation ,certaines variables soient astreintes à être entières ou même binaire $x_j = 0$ ou 1, on parle alors de programme linéaire discret "à variables entiers " ("Integer Linear Programming" noté "ILP") ou programme linéaire binaire ("Binary integer program" noté "BIP"). Un des problèmes le plus étudié dans la classe des (ILP) est le problème du voyageur de commerce ("The Travelling Salesman Problem" noté "TSP") qui consiste à la recherche d'un trajet minimal permettant à un voyageur de visiter n villes séparées par distances données en passant par chaque ville exactement une fois. Il commence par une ville quelconque et termine en retournant à la ville de départ. Quel chemin faut-il choisir afin de minimiser la distance parcourue ?

La notion de distance peut-être remplacée par d'autres notions comme le temps qu'il met ou l'argent qu'il dépense. En général, on cherche à minimiser le coût.

En tan que problème d'optimisation, le TSP est un problème NP-difficile. En effet, dans sa version symétrique, le nombre total de solution possibles est

$\frac{(n-1)!}{2}$, où n est le nombre de villes. Avec un telle complexité factorielle , une résolution efficace du TSP nécessite donc le recours à des méthodes d'optimisation très performants.

Les méthodes de résolution du TSP peuvent être réparti en deux groupes de nature diffèrante :

— La première groupe comprend les méthodes exactes qui garantissent la complétude de la résolution : c'est le cas de la méthode séparation et évaluation ("Branch and Bound") la méthode étudiée dans notre travail.

— Le second groupe comprend les méthodes approchées dont le but est de trouver une solution de bonne qualité en un temps de calcul raisonnable sans garantir l'optimalité de la solution obtenue.

Le manuscrit est réparti en quatre chapitres.

Au premier chapitre, nous donnons quelques rappels sur la théorie du graphe et de complexité.

Le deuxième chapitre est une introduction aux problème (ILP). L'accent sera mis sur le problème du TSP comme exemple de problème en variable binaire.

Au troisième chapitre, nous présentons les principes de base de la méthode de Branch and Bound.

Une variante adaptée au problème TSP (le cas pour l'algorithme de Little) est ensuite traitée. On a présenté ce principe par un exemple explicatif puis la méthode sera implémentée en langage c. Quelques tests numériques serons présentés par la suite.

Nous terminons notre travail par une conclusion et quelques références bibliographiques.

Chapitre 1

Notions sur les graphes

Dans ce chapitre, nous donnons quelques notions de théorie des graphes qui sont nécessaires pour introduire le probléme de voyageur de commerce [1].

1.1 Graphe

Définition 1. ***(Graphe)*** *De façon plus formelle, un graphe est défini par un couple* $G = (S, A)$ *tel que :*

- S *est un ensemble fini de sommets.*
- A *est un ensemble de couples de sommets* $(s_i, s_j) \in S_2$.

Un graphe peut être orienté ou non :

- *Dans* ***un graphe orienté****, les couples* $(s_i, s_j) \in A$ *sont orientés, c'est à dire que* (s_i, s_j) *est un couple ordonné, où* s_i *est le sommet initial, et* s_j *le sommet terminal. Un couple* (s_i, s_j) *est appelé* ***un arc****, et est représenté graphiquement par* $s_i \longrightarrow s_j$.
- *Dans* ***un graphe non orienté****, les couples* $(s_i, s_j) \in A$ *ne sont pas orientés, c'est à dire que* (s_i, s_j) *est équivalent à* (s_j, s_i)*. Une paire* (s_i, s_j) *est appelée* ***une arête****, et est représentée graphiquement par* $s_i - s_j$.

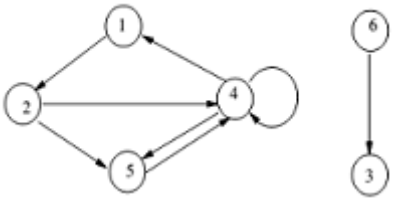


FIGURE 1.1 – Graphe orienté

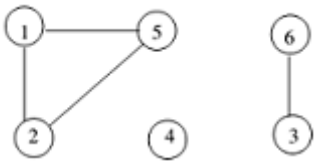
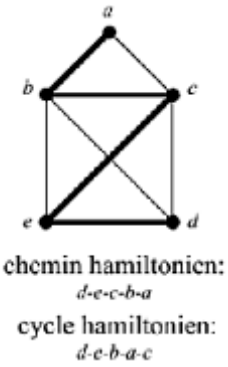


FIGURE 1.2 – Graphe non orienté

Définition 2. ***(Sommets adjacents)*** *Deux sommets qui sont reliés par une arête (orienté ou non) sont dits adjacents.*

Définition 3. ***(Chemin)*** *Un chemin est une suite de sommets reliés par des arêtes.*

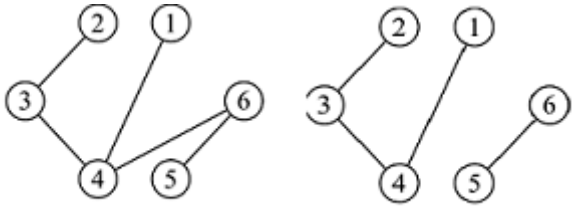
Définition 4. ***(Chemin hamiltonien)*** *Un chemin hamiltonien est un chemin dans le graphe qui passe par tous les sommets une et une seule fois. Si ce chemin est fermé (i.e. il existe une arête reliant le sommet de départ au sommet d'arrivée), on parlera de cycle hamiltonien.*



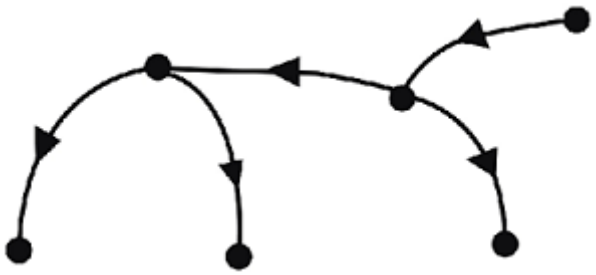
Définition 5. ***(Graphe connexe)*** *Un graphe est dit connexe si tout sommet est relié à tout autre sommet par une arête ou une suite d'arête.*

1.2 Arbres et arborescences

Définition 6. ***(Arbre)*** *Un arbre est un graphe connexe sans cycles. Un graphe sans cycle qui n'est pas connexe est appelé une forêt (chaque composante connexe est un arbre).*



Définition 7. ***(Arborescence)*** *Un graphe G est une arborescence s'il existe un sommet R appelé racine de G tel que, pour tout sommet S de G, il existe un chemin et un seul de R vers S.*



Arborescence

1.3 Notions sur la complexité

Définition 8. *Une fonction $f(n)$ est $O(g(n))$ ($f(n)$ est de complexité $g(n)$), s'il existe un réel $c > 0$ et un entier positif n_0 tel que pour tout $n \geq n_0$ on a $|f(n)| \leq c.g(n.)$*

n	$\lg n$	$n^{0.5}$	n^2	2^n	$n!$
10	3.32	3.16	10^2	$1.02 * 10^3$	$3.6 * 10^6$
100	6.64	10.00	10^4	$1.27 * 10^{30}$	$9.33 * 10^{157}$
1000	9.97	31.62	10^6	$1.07 * 10^{301}$	$4.02 * 10^{2567}$

Table 1.1 : Des typique fonction

Définition 9. *Un algorithme en temps polynomial est un algorithme dont le temps de la complexité est en $O(p(n))$, où p est une fonction polynomiale et n est la taille de l'instance (ou sa longueur d'entrée). Si k est le plus grand exposant de ce polynôme en n, le problème correspondant est dit : résoluble en $O(n^k)$ et appartient à la classe P, un exemple de problème polynomial est celui de la connexité dans un graphe.*

Définition 10. ***(La classe NP)*** *La classe NP contient les problèmes de décision qui peuvent être décidés sur une machine non déterministe en temps polynomial. C'est la classe des problèmes qui admettent un algorithme polynomial capable de tester la validité d'une solution du problème. Intuitivement, les problèmes de cette classe sont les problèmes qui peuvent être résolus en énumérant l'ensemble de solutions possibles et en les testant à l'aide d'un algorithme polynomial.*

Définition 11. ***(La classe NP-complet)*** *Parmi l'ensemble des problèmes appartenant à NP, il en existe un sous ensemble qui contient les problèmes les plus difficiles : on les appelle les problèmes NP-complets. Un problème NP-complet possède la propriété que tout problème dans NP peut être transformé (réduit) en celui-ci en temps polynomial. C'est à dire qu'un problème est NP-complet quand tous les problèmes appartenant à NP lui sont réductibles. Si on trouve un algorithme polynomial pour un problème*

NP-complet, on trouve alors automatiquement une résolution polynomiale de tous les problèmes de la classe NP.

Définition 12. ***(La classe NP-difficile)*** *Un problème est NP-difficile si savoir le résoudre en temps polynomial impliquerait que l'on sait résoudre en problème (de décision) NP-complet en temps polynomial. Les problème NP-difficiles sont donc dans sens plus dur que les problème NP-complet.*

Par exemple :
Le problème de voyageur de commerce est NP-difficile.
La preuve de ce résultat découle de la NP-complétude du problème de cycle hamiltonien.

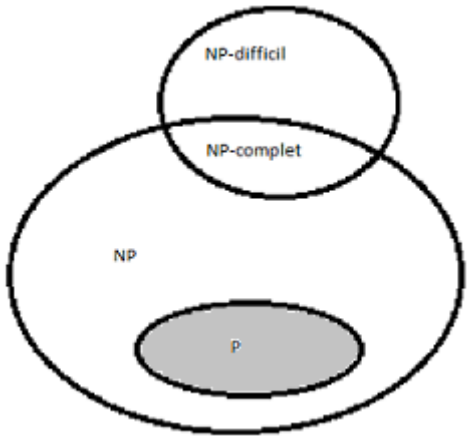


FIGURE 1.3 – Les problèmes NP.

Chapitre 2

Programmation linéaire en nombres entières

Nous abordons dans cette chapitre une nouvelle catégorie de problèmes : les problèmes de programmation linéaire en nombres entiers. Nous nous intéresserons essentiellement à la modélisation de ces problèmes et non à leur résolution et présenterons quelques-uns des très nombreux problèmes susceptibles d'être modélisés par un problème de programmation linéaire en nombres entiers [11].

2.1 Définition d'un ILP

2.1.1 Formulation d'un ILP

Un programme linéaire (*linear programming"LP"*) en variables continues est formulé :

$$(LP)\begin{cases} z & = min\, c^T x \\ & Ax \leq b, \\ & x \geq 0, \end{cases}$$

avec $A \in \mathbb{R}^{m\times n}$, le second membre $b \in \mathbb{R}^{m\times 1}$ et $c \in \mathbb{R}^{n\times 1}$. Lorsque toutes les variables doivent être entières, le problème résultant noté *(ILP)* (*" integer linear programming"*) est le problème général de la programmation linéaire en variables entières :

$$(ILP)\begin{cases} z &= min\, c^T x \\ & Ax \leq b, \\ & x \in \mathbb{N}^n. \end{cases}$$

Si les varaibles $x \in \{0,1\}^n$, on dit qu'on a un programmation linéaire en variables binaires noté *(BIP)* ("*Binary integer program*")

$$(BIP)\begin{cases} z &= min\, c^T x \\ & Ax \leq b, \\ & x \in \{0,1\}^n. \end{cases}$$

Si une partie seulement des variables doivent être entières, le problème résultant est un problème de programmation linéaire mixte en variables entières et varaibles réelles, noté *(MILP)* ("*mixed integer linear programming*"). En séparant les deux types de varaibles dans la fonction objectif et les contraintes, un *(MILP)* se formule :

$$(MILP)\begin{cases} z &= min\, c_1^T x + c_2^T y \\ & A_1 x + A_2 y \leq b \\ & x \in \mathbb{N}^p, x \in \mathbb{R}_+^{n-p}. \end{cases}$$

Où $A_1 \in \mathbb{R}^{m\times p}$, $A_2 \in \mathbb{R}^{m\times(n-p)}$, $c_1 \in \mathbb{R}^{p\times 1}$, $c_2 \in \mathbb{R}^{(n-p)\times 1}$ et le second membre $b \in \mathbb{R}^{m\times 1}$. La question de résoudre efficacement un *(ILP)* se pose, on sait résoudre efficacement un *(LP)*. En théorie, le problème est de complexité polynômiale et en pratique, bien que exponentiel dans le pire des cas le simplexe fonctionne bien. Peut-on obtenir les mêmes résultats pour le problème *(ILP)* ?

2.1.2 Relaxation linéaire continue

Le problème *(LP)* obtenu par relaxation des contraintes d'intégrité d'un *(ILP)* définit une relaxation de ce *(ILP)*. Pour un probléme de minimisation, la solution optimale du *(LP)* donne une borne inférieure de la valeur de la solution optimale du *(ILP)*, il s'agit d'un relachement de la contrainte $x \in P \cap \mathbb{N}^n$ en $x \in P$ telle que

$$P = \{x \in \mathbb{R}^n_+ : Ax \leq b\}. \tag{2.1}$$

Comme $P \cap \mathbb{N}^n \subseteq P$ et la fonction objective ne change pas alors il est clair qu'il est une relaxation [1] .

La relaxation linéaire ne donne pas seulement un borne inférieure mais des fois preuve l'optimalité.

Exemple 1. *Le programme de la relaxation linéaire d'un problème ILP suivant :*

$$\begin{cases} z & = \max 7x_1 + 4x_2 + 5x_3 + 2x_4 \\ & 3x_1 + 3x_2 + 4x_3 + 2x_4 \leq 6 \\ & x_i \in \{0,1\} \qquad i = 1, \ldots, 4 \end{cases}$$

a une solution optimale $x^* = (1, 1, 0, 0)$

Comme les x_i sont entier , alors le probleme *(ILP)* associe a la mème solution.

L'exemple suivant nous dit que la solution du *(RLP)* n'a parfois rien à voir avec la solution *(ILP)*.

Exemple 2. *Considérons le programme linéaire en nombre entier suivante :*

$$\begin{cases} z & = \max 10x_1 + 11x_2 \\ & 10x_1 + 12x_2 \leq 59 \\ & x_1, x_2 \in Z \end{cases}$$

1. Relaxer : relâcher des contraintes sans dégrader les solutions

En varaibles continues, la solution optimale est le point $x_1 = 5.9$, $x_2 = 0$, $(z^{LP} = 59)$ *qui n'est pas entière.*

Une simple méthode d'arrondi conduit à $x_1 = 6$, $x_2 = 0$, $(z = 60)$ *qui n'est pas admissible pour (ILP) !*

2.2 Problèmes linéaires à solutions entières

Certains problèmes à coefficients entiers ont naturellement des solutions entières.

Définition 13. *Soit* B *une matrice carrée d'ordre* n, B *est dite* ***unimodulaire (UM)*** *si* $\det(B) \in \{0, 1, -1\}$.

Définition 14. *Une matrice* A *de taille* $m \times n$ *est dite* ***totalement unimodulaire (TUM)*** *si toute sous-matrice carrée et non singulière de* A *est* ***UM***.

Remarque 1. *Toute matrice totalement unimodulaire est nécessairement composée de* $0, +1$ *ou* -1.

Considérons le problème *(LP)* donne sous la forme standard $i.e$,

$$(LP) \begin{cases} z & = min\, c^T x \\ & x \in D \end{cases}$$

Avec $D = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$. Nous avons démonter le théorème suivant

Théorème 1. *Si* A *est* ***TUM****, alors tous les sommets de* **D** *sont entiers pour tout vecteur entier* b.

Donc, la résolution par le simplexe d'un *(ILP)* standard relaxé dont la matrice est **TUM** donne la solution entiere exacte de *(ILP)*.

Dans le cas de contraintes inégaliter *i.e* $x \in \mathbf{P}$, où $\mathbf{P}$ donné par la relation (2.1). Nous avons le théorème suivant :

Théorème 2. *Si A est **TUM**, alors tous les sommets de P sont entièrs pour tout vecteur entier b.*

Preuve 1. *Il revient donc à montrer que si A est **TUM**, alors (A/I) l'est aussi. Nous ajoutons des variables d'écart et on applique le théorème (1). Soit C une sous-matrice carré inversible de (A/I). Les lignes de C peuvent être permutés afin de pouvoir écrire*

$$C = \begin{pmatrix} B & 0 \\ L & I_k \end{pmatrix}$$

*où I_k est une matrice identité de taille k et B est une sous-matrice carrée de A. On a $\det(C) = \det(B) = \pm 1$ parce que A est **TUM** et C est inversible.*

Remarque 2. *La condition d'unimodularité est suffisante mais non nécessaire à l'existence d'une solution optimale entière du problème (LP). En pratique, de nombreux problèmes d'optimisation à données entières ont une solution entière bien que la matrice A ne soit pas unimodulaire, mais l'intégrité de la solution risque d'être perdue si des paramètres changent.*

Propriété 1. ***(Condition suffisante)** Soit A une matrice contenant seulement les éléments $0, +1$ ou -1 et qui satisfait les deux conditions suivantes*

1. *Chaque colonne contient au plus deux éléments non-nuls.*
2. *Les lignes de A peuvent être partitionnées en deux sous-ensembles I_1 et I_2 tels que pour chaque colonne contenant deux éléments non-nuls :*
   - *si les deux éléments non-nuls ont le même signe alors l'un est dans I_1 et l'autre dans I_2.*
   - *si les deux éléments non-nuls sont de signes différents alors ils sont tous les deux dans I_1 ou tous les deux dans I_2.*

*Alors A est **totalement unimodulaire**.*

2.3 Exemples

La programmation linéaire en variables entières est très utile comme langage de modélisation, car la plupart des problèmes réels comportent des variables qui doivent, par nature, prendre nécessairement une valeur entière. Nous donnerons ici deux exemples pratique de *(ILP)*.

2.3.1 Problème d'affectation

Le problème d'affectation (*"Assignment problem"*, *AP*) est un problème particulier de *(LP)* en varaibles binaires, il consiste à affecter au mieux des tâches à des agents. Chaque agent peut réaliser une unique tâche pour un coût donné et chaque tâche doit être réaliser par un unique agent. Les affectations (c'est à dire les couples agent-tâche) ont toutes un coût défini.

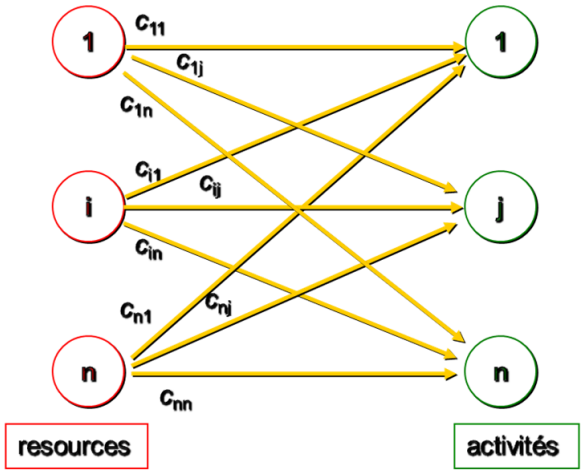


FIGURE 2.1 – Representation graphique de AP.

Le but étant de minimiser le coût total des affectations afin de réaliser toutes les tâches. Le problème peut être énoncé de la manière suivant :

Désignons par $i = 1, 2, ..., n$ les agents, $j = 1, 2, ..., n$ les tâches et introduisons

les n^2 varaibles binaires :

$$x_{ij} = \begin{cases} 1 & \text{si l'agent } i \text{ fait la tâche, } j \\ 0 & \text{sinon.} \end{cases}$$

Les contraintes du problème d'affectation s'écrivent donc simplemnet :

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{pour } i = 1, \ldots, n.$$

Chaque agent i fait un seule tâche j

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \text{pour } j = 1, \ldots, n,$$

Chaque tâche j elle faire par un seule agent i.

Définition de la fonction objectif : Nous traiterons ici le problème linéaire d'affectation. Soit c_{ij} le coût que l'agent i fait la tâche j donc la fonction objectif revient le minimum des coûts :

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

Le problème linéaire d'affectation s'écrit donc :

$$(AP) \begin{cases} \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\ \sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n \\ \sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1, \ldots, n \\ x_{ij} \in \{0, 1\} \qquad i = 1, \ldots, n, \qquad j = 1, \ldots, n \end{cases}$$

2.3.2 Problème du voyageur de commerce

Historique

Les premières approches mathématiques exposées pour le problème du voyageur de commerce ont été traitées au 19 siècle par les mathématiciens Sir William Rowan Hamilton et Thomas Penyngton Kirkman.

Hamilton en a fait un jeu : " Hamilton's Icosian game " . Les joueurs devaient réaliser une tournée passant par 20 points en utilisant uniquement les connections prédéfinies.

En 1930, ce problème est traité plus en profondeur par Karl Menger à Harvard. Il est ensuite développé à Princeton par les mathématiciens Hassler Whitney et Merril Flood, une attention particulière est portée sur les connections par Menger et Whitney ainsi que sur la croissance de ce problème.

En 1954, des solution de ce problème pour 49 villes ont été obtenues par Dantzig, Fulkerson et Johnson par une méthode de coupe.

En 1975, des Solution pour 100 villes par Camerini, Fratta and Maffioli.

En 1987, le problème à été résolu pour 532 et 2392 villes par Padberg et Rinaldi.

En 1998, des solution ont été trouvées pour 13 509 villes des Etats-Unis.

En 2001, Applegate, Bixby, Chvàtal et Cook des universités de Rice et Princeton ont résolu le problème pour les 15 112 villes d'Allemagne.

En mai 2004, le problème du voyageur de commerce qui consiste à visiter chacune des 24.978 villes en Suède a été résolu :

Une excursion de longueur approximativement de 72.500 kilomètres a été trouvée et on a montrée l'inéxistence d'une tournée plus courte.

En mars 2005, un problème de taille égale 33.810 villes a été résolu (Cook et al. 2006).

Position du probléme TSP

Le problème du voyageur de commerce (" *Traveling Salesman Problem, TSP*) nait d'une problématique vécue par des vendeurs ou commerciaux qui déplacent pour livrer ou rencontrer leurs clients. C'est l'un des problèmes le plus étudié dans l'optimisation combinatoires [2] qui à la recherche d'un trajet minimal permettant à un voyageur de visiter n villes séparées par distances données en passant par chaque ville exactement une fois. Il commence par une ville quelconque et termine en retournant à la ville de départ. Quel chemin faut-il choisir afin de minimiser la distance parcourue ?

Etant donné un graphe $G = (V, E)$ non orienté simple et sans boucles, le problème du voyageur de commerce consiste à déterminer un cycle hamiltonien (qu'on appelera tournée) de longueur minimale.

Formulation mathématique du problème

Le *TSP* peut se modéliser comme *(BIP)*. En effet, étant donnée n villes, notons $C = (c_{ij})$ la matrice des coûts et x_{ij} les varaibles de décision définies par

$$x_{ij} = \begin{cases} 1 & \text{si le voyageur va immédiatement de la ville i vers la ville j,} \\ 0 & \text{sinon ;} \end{cases}$$

La formulation mathématique est

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{2.2}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{pour } i = 1, \ldots, n \tag{2.3}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \text{pour } j = 1, \ldots, n \tag{2.4}$$

2. Programme linéaire discret " à varaibles entières" binaires "0 ou 1"

$$\sum_{i \in S} \sum_{j \in \overline{S}} x_{ij} \geq 1, \quad S \subset V, \qquad 2 \leq \mid S \mid \leq n-2 \tag{2.5}$$

$$x_{ij} \in \{0,1\} \text{ pour } i,j = 1,\ldots,n \qquad i \neq j \tag{2.6}$$

Dans cette formulation, la relation (2.2) décrit la fonction objectif. Les contraintes (2.3) et (2.4) s'appellent les contraintes de degré qui assurent qu'une ville visitée qu'une seule fois : on y arrive une et une seule fois (2.3), on en part une et une seule fois (2.4), ces contraintes ne sont pas suffisantes pour décrire les tours, d'où la nécessité d'introduire les contraintes (2.5) appelées contraintes d'élimination des sous-tours, avec S un sous ensemble de V et $\overline{S}$ son complémentaire dans V, $|S|$ est le cardinal de S. Enfin, les contraintes (2.6) sont les contraintes d'intégrité de varaibles.

Une autre formulation des contraintes (2.5) peut être donnée par la relation suivante

$$\sum_{i,j \in S} x_{ij} \leq \mid S \mid -1, \qquad S \subset V, \qquad 2 \leq \mid S \mid \leq n-1 \tag{2.7}$$

Cette formulation de *TSP* contient $n(n-1)$ varaibles binaires, $2n$ contraites de degré et $2^n - 2n - 2$ contraintes d'élimination de sous tours.

2.4 Méthodes de résolution

La première idée qui vient à l'esprit lorsque l'on veut résoudre TSP est certainement celle de la recherche exhaustive. Le principe en est très simple, mais au prix d'une complexité en temps très élevée : il consiste à déterminer toutes les solutions, à en évaluer la valeur, puis à sélectionner la meilleure de ces solutions. Dans notre contexte, cela se traduit par la recherche de tous les cycles hamiltoniens. Or, dans un graphe complet K_n, il y a $\frac{(n-1)!}{2}$ tours possibles ; sachant que l'évaluation d'un tour nécessite un temps $O(n)$, nous obtenons une complexité totale en temps de $O(n!)$. Par exemple , si le calcul d'un chemin prend une microseconde , alors le calcul de tous les chemins pour 10 points est de 181440 microsecondes soit 0.18 seconde mais pour 15 points, cela représente déjà 43589145600 microsecondes soit un peu plus de 12 heures et pour 20 points de $6 * 1016$ microsecondes soit presque deux millénaires (1927 années).

Il existe trois grandes catégories de méthodes de résolution des problémes (BIP) : les méthodes exactes, les méthodes heuristiques et les méthodes métaheuristiques [9].

Les méthodes exactes permettent d'obtenir une solution optimale à chaque fois, mais leurs durées de calcul tendent à augmenter exponentiellement avec la taille du problème. A l'inverse, les méthodes heuristiques sont des méthodes spécifiques, permettant d'obtenir rapidement une solution approchée de bonne qualité, mais qui n'est donc pas nécessairement optimale. Les plus utilisées sont celle du " plus proche voisin " et les " méthodes d'insertion. Les méthodes métaheuristiques sont des algorithmes d'optimisation généralement de type stochastique combinant plusieurs approches heuristiques. Les métaheuristiques sont souvent inspirées par systèmes naturals, qu'ils soient pris en physique (cas du recuit simulé), en biologie de l'évolution (cas des algorithmes génétiques) ou encore en éthologie (cas des algorithmes de colonies de fourmis). les deux

types de méthodes heuristiques et metaheuristique ne sera pas abordé dans ce mémoire.

Pour le probléme *TSP*, l'une des méthodes exactes les plus clasiques et les plus performantes reste la procédure par séparation et évaluation (" Branch and Bound "). Cette méthode est présenteé dans le chapitre suivant.

Chapitre 3

Méthode Branch and Bound

Ce chapitre est consacré à la méthode par séparation et évaluation (Branch and Bound) [11]. Les méthodes par séparation et évaluation sont le moyen générique le plus utilisé pour la résolution exacte des problèmes d'optimisation combinatoire et en particulier pour la résolution des *(ILP)* qui pratiquent une énumération intelligente de l'espace des solutions.

3.1 Présentation de l'algorithme "Branch and Bound"

Soit le problème en varaibles discrète :

$$(P)\begin{cases} z & = min\, F(x) \\ & x \in S \subset \mathbb{N}^n. \end{cases}$$

Où S contient un nombre fini de solutions. $F(x)$ est la fonction objective.

L'algorithme consiste à séparer de manière récursive le problème en sous problèmes de cardinalité inférieure tant que la résolution de ces problèmes reste difficile. Le cardinal de l'ensemble à explorer est réduit en imposant à cet ensemble des contraintes supplémentaires (réduction du domaine). Une série de

tests, appliqée à tous les sous problèmes permet de supprimer de l'espace de recherche les sous problèmes qui ne peuvent pas engendrer de solution optimale. Cette recherche par décomposition de l'ensemble des solutions peut être représentée graphiquement par un arbre. C'est de cette représentation que vient le nom de " méthode de recherche arborescence [1]".

Une méthode de branch and bound est basé sur de trois éléments principaux :

— La procédure de séparation.

— La procédure d'évaluation.

— La stratégie de parcours ou procédure de cheminement.

3.1.1 Procédure de séparation ("Branching process")

Séparer (brancher) de manière récursive le problème en sous problèmes de cardinalité inférieure tels que l'union de leurs espaces de solutions forme l'espace des solutions du problème pére, le noeud séparé en priorité est celui qui produit la borne inférieure la plus faible. Un sous ensemble qui ne peut être séparé est appelé ensemble sondé.

Exemple 3. *Si* $S \subseteq \{0,1\}^3$ *on peut construire l'énumération suivante :*
D'abord on divise S *en deux ensembles :*
$S_0 = \{x \in S; x_1 = 0\}$ *et* $S_1 = \{x \in S; x_1 = 1\}$ *,*
en suite , $S_{00} = \{x \in S_0 : x_2 = 0\} = \{x \in S; x_1 = x_2 = 0\}, S_{01} = \{x \in S_0; x_2 = 1\}$
ainsi de suite

1. Une arborescence dite arbre de décision : chaque noeud de l'arbre est un problème.

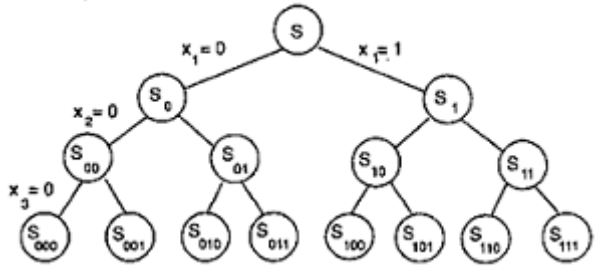


FIGURE 3.1 – Décomposition d'un problème binaire

Exemple 4. *Un autre exemple est le problème d'énumération de tous les tours d'un problème de voyageur de commerce (traveling salesman problem)*

D'abord on divise S l'ensemble de tous les tours de 4 villes à S_{12}, S_{13}, S_{14} où S_{ij} est l'ensemble de tous les tours qu'ils contenent l'arc (i, j) , ensuite S_{12} est divisé à $S_{(12)(23)}$ et $S_{(12)(24)}$ et ainsi de suite

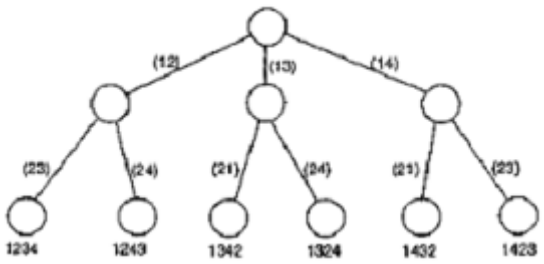


FIGURE 3.2 – Décomposition le problème TSP

Le principe de séparation peut , par exemple, être basé sur une simple idée est choisis un entier qu'il est une partie de la solution de problème linéaire relaxé et on divise le problème à deux sous problème séparé par ce point :

Si $x_j = \bar{x}_j \notin \mathbb{Z}$ alors un peut être choisis parmi :

$$S_1 = S \cap \{x : x_j \leq [\bar{x}_j]\}, \qquad S_2 = S \cap \{x : x_j \geq [\bar{x}_j] + 1\}.$$

Il est claire que $S = S_1 \cup S_2$ et $S_1 \cap S_2 = \emptyset$. Un notre cause pour cette choix est que la solution $\bar{x}$ de $LP(S)$ n'est pas un solution admissible de $LP(S_1)$

ou $LP(S_2)$ ce implique s'il n'existe pas un multiple des solution optimales de (LP) alors

$$\max\{z^{LP(S_1)}, z^{LP(S_2)}\} < z^{LP(S)}$$

donc la borne supérieure va décroissante strictement.

3.1.2 Procédure d'évaluation ("Bounding process") et de sondage

Définition 15. *Une fonction d'évaluation est une fonction $v : S^i \to v_i$ qui associe une valeur v_i à chaque sous- ensemble S^i.*

Une procédure d'évaluation qui consiste à analyser un sous- problème : l'essentiel, cette analyse vise à évaluer la valeur optimale de la fonction objectif de sous-problème, plus précisement à déterminer une borne inférieure de cette valeur.

Le Branch and Bound utilise, à des fins d'élagage [2], deux fonction :

Une borne inférieure de la fonction d'utilité du problème initaile, qui résulte d'une fonction d'évaluation.

et **Une borne supérieure** de la fonction d'utilité des solutions d'un sous ensemble.

La connaissance d'une borne inférieure du problème et d'une borne supérieure de la fonction d'utilité de chaque sous ensemble permet de stopper l'exploration d'un sous ensemble de solution ne pouvant pas contenir de solutions condidates à l'optimalité. Si pour un sous problème la borne supérieure est plus petite que la borne inférieure du problème, l'exploration du sous ensemble correspodant est inutile. D'autre part, lorsque le sous ensemble est suffisament petit, on procède à une évaluation dite exacte : on résout alors le sous problème

2. Élimination de branches dans l'arborescence de recherche

correspondant. La valeur v_i doit être une inférieure de la valeur optimale de la fonction objectif du problème P^i :

$$v_i \leq \widehat{F}_i = \min F(x) \qquad x \in S^i.$$

En cas de maximisation, il s'agit d'une règle de majoration :

$$v_i \geq \widehat{F}_i = \max F(x) \qquad x \in S^i$$

Remarque 3. *Soient $S^{(ik)}$, $k = 1, ..., p$ les sous-ensembles obtenus par séparation du sous ensemble S^i. Etant donné $S^{(ik)} \subset S^{(i)}$, il vient $\widehat{F}_i \leq \widehat{F}_{ik} \quad \forall k$ et par conséquent $v_i \leq \widehat{F}_{ik} \quad \forall k$.*

Ainsi la valeur de la fonction d'évalution au noeud parent, constite déjà une borne inférieure de la valeur optimale de la fonction économique pour tous les sous-noeuds qui en sont issue. Dès lors $v_i \leq v_{ik} \quad \forall k$.

C'est-à- dire que la fonction d'évaluation est une fonction non décroissant (non croissante pour une maximixation) lorsque une nouvelle séparation est éffféctuée.

Détermination d'une fonction d'évaluation

Pour obtenire la borne inférieure v_i de $\widehat{F}_i$, la technique la plus utilisée est de procéder à une relaxation du problème $P^{(i)}$. Une telle relaxation consiste à élargir l'ensemble $S^{(i)}$ en un ensemble $R^{(i)}$ tel que $S^{(i)} \subset R^{(i)}$ et résoudre le problème relaxé $(RP^{(i)})$:

$$(RP^{(i)}) \begin{cases} \widehat{F}_i^R & = min\, F(x) \\ & x \in R^{(i)}, \end{cases}$$

de sort que $v_i = F_i^R \leq F_i$.

Les technique de relaxation sont souvent utilisées dans le calcul d'une borne inférieure. Voici dans ce qui suit, les principales techniques de relaxation :

a) **Relaxation linéaire :**

Déjà introduite au pragraphe (2.1.2).

b) **Relaxation Lagrangienne :**

La relaxaction lagrangienne s'articule sur l'idée de relâcher les contraintes difficiles, non pas en les supprimant totalement, mais en les prenant en compte dans la fonction objectif de sorte qu'elles pénalisent la valeur des solution qui les violent.

La relaxation Lagrangienne est appliquée en générale lorsqu'on reconnaît dans la matrice des contraintes, des contraintes difficilles "$Dx \leq d$ " dont la relaxation engendrera des problèmes plus faciles à resoudre, ou bien pour lesquels on dispose d'outils efficaces de résolution .

Les contraintes [3] relâchées sont réinjectées dans la fonction objectif, ponderées par les coefficients $\lambda = (\lambda_1, \ldots, \lambda_m)^T \in \mathbb{R}_+^m$ appellés multiplicateurs .

Considérons le problème (ILP) et notons $X = \{x \in \mathbb{Z}_+^n \mid Ax \leq b\}$, on définit la fonction Lagrangienne comme suit :

$$L(x, \lambda) = c^T x + \lambda^T (Dx - d)$$

On définit la fonction dual :

$$(L_\lambda) \begin{cases} d(\lambda) & = minL(x, \lambda) = min(c^T x + \lambda^T (Dx - d)) \\ & x \in X \end{cases}$$

c) **Relaxation des contraintes :**

Une technique simple de relaxation consiste à ignorer certaines contraintes du problème.

3. Lorsque les m contraintes qui sont dualisée sont des contraintes d'égalité de la forme "$Dx = b$", les multiplicateurs de Lagrange correspondant sont de signe quelconque ($\lambda \in \mathbb{R}^m$)

On obtient alors un problème dont la solution optimale est plus facile à calculer. Par exemple :
Soit un polyèdre $D^{(k)}$ défini par deux ensembles de contraintes :

$$D^{(k)} = \{x \backslash A_1 x \leq b_1, A_2 x \leq b_2\}.$$

Il peut s'avérer que la suppression d'un des deux ensembles de contraintes, par exemple le second permet d'obtenir un polyèdre convexe,

$$D'^{(k)} = \{x \backslash A_1 x \leq b_1\},$$

tel que le problème relaxé ($RP^{(i)}$) défini par $R^{(i)} = D'^{(k)} \cap \mathbb{Z}_+^n$ est un problème classique, évident ou simple à résoudre. Le problème relaxé ($RP^{(i)}$) est alors appelé sous problème du problème ($P^{(i)}$) et la fonction d'évaluation est obtenue par résolution du sous-problème.

Conditions(tests) de sondage :

Plus généralement un sous-ensemble ($S^{(i)}$) sera sondé si une des trois conditions suivantes est réalisée :

α) $R^{(i)} = \emptyset$: dans ce cas, a fotiori, $S^{(i)} = \emptyset$

β) La solution optimale du problème relaxé ($RP^{(i)}$) appartienne à $S^{(i)}$:

- Il s'agit de la solution optimale du problème $P^{(i)}$ (et $v_i = \tilde{f}_i$).
- si de plus $\tilde{f}_i \leq \tilde{f}$, il convient d'actualiser $\hat{x}$ et $\hat{f}$.

γ) $\hat{f} \leq v_i$: dans ce cas $f(\hat{x}) \leq f(x) \qquad \forall x \in S^{(i)}$

Exemple 5. *Dans la figure (3.3) suivante on a une décomposition d'un ensemble S a deux sous ensembles S_1 et S_2 avec ces bornes supérieure et inférieure corespondant :*

On observe que la borne supérieure et la borne inférieure sont égaux, $UB^{(S1)} = LB^{(S1)} = 20$ donc ne peut pas déviser S_1 donc la branche S_1 est éliminé par optimalité.

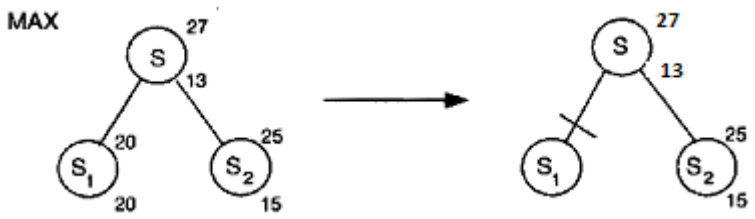


FIGURE 3.3 – Éliminer par optimalité.

Exemple 6. *Dans la figure (3.4) suivante on a une autre décomposition d'un ensemble S a deux sous ensembles S_1 et S_2 avec ces bornes supérieure et inférieure corespondant :*

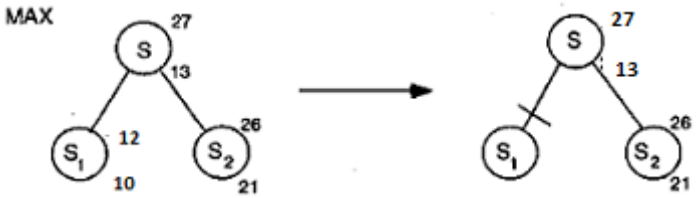


FIGURE 3.4 – Éliminer par borne.

On observe que la valeur optimale est minoré par 21 *, mais la borne supérieur* $UB^{(S1)} = 20$ *donc l'ensemble* S_1 *ne contient pas la solution optimale, donc on éliminé la branche* S_1 *a partir de borne.*

Exemple 7. *Dans la figure (3.5) suivante on a une autre décomposition d'un ensemble S a deux sous ensembles S_1 et S_2 :*

On a : $UB = \max\{24, 37\} = 37$ *et* $LB = \max\{13, -\} =?$ *ne peut pas conclure.*

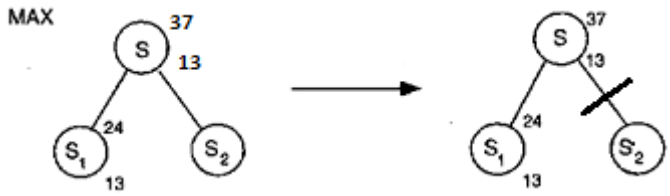


FIGURE 3.5 – Éliminer par infaisabilité.

A partir a ces exemples on remarque une liste des raisons pour que on éliminé une branche. Le nœud i de l'arboscence pourra être sondé par application des règle générales de sondage c'est -à-dire si soit :

- Éliminé par optimalité,
- Éliminé par borne,
- Éliminé par infaisabilité : $S_t = \emptyset$.

3.1.3 Procedure de cheminement (la stratégie de parcours)

Lorsqu'un nœud de l'arborescence est sondé, il conviendra de remonter dans l'arborescence vers un autre nœud situé à un niveau inférieur ou égal dans l'arborescence ; cette partie de la procédure de cheminement est souvent appelée "**backtracking process**". Il est évidement exclu d'examiner la totalité des nœuds de l'arborescence. Pour réaliser une énumération implicite efficace, il convient de définir le plus adéquatement possible la façon de cheminer dans l'arborescence. Parmi les stratégies de parcours les plus connues :

a) Procédure en profondeur d'abord [3] : Parfois dénommée "Procédure par séparation et évaluation séquentielle (PSES)" :
L'idée de cette procédure de cheminement est bien sûr de descendre rapidement dans l'arborescence- sans examiner trop de nœuds et sans faire trop d'analyses- vers un nœud dont on espère qu'il contient une trés

bonne solution. Si c'est le cas, l'obtention de celle-ci permettra de sonder beaucoup de nœuds - grâce au test de borne - lors du processus de remontée dans l'arborescence.

b) Procédure en largeur d'abord[3] :
On parle de procédure "en largeur d'abord" si systématiquement tous les nœuds d'un même niveau de l'arborescence sont examinés avant de considérer le niveau suivant.

c) Procédure la meilleure évaluation d'abord[3] : Parfois dénommée "Procédure par séparation et évaluation progressive" (PSEP) :
Cette stratégie favoris l'exploration des sous problèmes possédant la plus grande borne supérieure. Elle dirige la recherche là où la probabilité de trouver une meilleure solution est la plus grand. Elle permet aussi d'éviter l'exploration de tous les sous problèmes qui possédent une évalution inférieure à la valeur optimale.

3.1.4 La méthode Branch and Bound sur un exemple

La méthode de branch and bound (procédure par évaluation et séparation progressive) consiste à énumérer ces solutions d'une manière intelligente en ce sens que, en utilisant certaines propriétés du problème en question, cette technique arrive à éliminer des solutions partielles qui ne mènent pas à la solution que l'on recherche.

De ce fait, on arrive souvent à obtenir la solution recherchée en des temps raisonnables. Bien entendu, dans le pire cas, on retombe toujours sur l'élimination explicite de toutes les solutions du problème.

Pour ce faire, cette méthode se dote d'une fonction qui permet de mettre une borne sur certaines solutions pour soit les exclure soit les maintenir comme des solutions potentielles.

La caractéristique principale de cette technique consiste dans le fait que l'algorithme se propose de trouver dans quelle branche les solutions ont le plus de chance d'être trouvées pour ne pas continuer inutilement et voila un exemple pour comprendre.

Exemple 8. *Soit le programme en nombre entier suinant :*

$$(ILP)\begin{cases} z^{(ILP)} = & \max 4x_1 - x_2 \\ & 7x_1 - 2x_2 \leq 14 \\ & x_2 \leq 3 \\ & 2x_1 - 2x_2 \leq 3 \\ & x_1, x_2 \in \mathbb{N}, \end{cases}$$

on ajoute les variables d'écart et on résout le problème relaxé, on obtient :

$$\begin{cases} UB = & \max 59/7 - 4/7x_3 - 1/2x_4 \\ & x_1 - 1/7x_3 + 2/7x_4 = 20/7 \\ & x_2 + x_4 = 3 \\ & -2/7x_3 + 10/7x_4 + x_5 = 23/7 \\ & x_1, x_2, x_3, x_4, x_5 \geq 0, \end{cases}$$

on obtient alors un borne supérieur $UB = 59/7$ et un solution réel non entier $(\bar{x_1}, \bar{x_2}) = (\frac{2}{7}, 3)$.

Existe-il une méthode fixée pour trouver une solution admissible (qui donne le borne inférieur) ? [4].

Comme $LB < UB$ on divise ou branche. A partir de cette idée , on a $\bar{x_1} \notin \mathbb{N}$, on prend :

$$S_1 = S \cap \{x : x_1 \leq 2\} \text{ et } S_2 = S \cap \{x : x_1 \geq 3\}.$$

4. En générale non, par convention quand la solution admissible n'est pas évidente, la borne inférieur est $LB = -\infty$

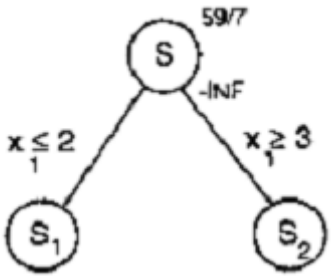


FIGURE 3.6 – Une partielle branche 1.

On a donc les sous problème suivant dans la figure (3.6). La liste des noeuds est S_1, S_2, donc on choisis arbitrarement S_1. La résolution des nouveau programme linéaire modifiées $LP(S_i)$ pour $i = 1, 2$:

On utilise l'algorithme dual simplexe [3], typéquement a partir de quelque pivots on obtient la solution optimale, on applique ça sur le programme linéaire $LP(S_1)$, on peut écrire la nouvelle contrainte $x_1 \leq 2$ i.e $x_1 + s = 2, s \geq 0$, ou l'écriture avec les variables d'écarts devient :

$$-\frac{1}{7}x_3 - \frac{2}{7}x_4 + s = -\frac{6}{7}.$$

Ça nous donne le problème :

$$\begin{cases} UB_{S1} = & \max \frac{59}{7} - \frac{4}{7}x_3 - \frac{1}{7}x_4 \\ & x_1 + \frac{1}{7}x_3 + \frac{2}{7}x_4 = \frac{20}{7} \\ & x_2 + x_4 = 3 \\ & -\frac{2}{7}x_3 + \frac{10}{7}x_4 + x_5 = \frac{23}{7} \\ & -\frac{1}{7}x_3 - \frac{2}{7}x_4 + s = -\frac{6}{7} \\ & x_1, x_2, x_3, x_4, x_5, s \geq 0, \end{cases}$$

après deux pivots de simplexe on obtient :

$$\begin{cases} UB_{S1} = & \max \frac{15}{2} - \frac{1}{2}x_5 \\ & x_1 + s = 2 \\ & x_2 - \frac{1}{2}x_5 + s = \frac{1}{2} \\ & x_3 - x_5 - 5s = 1 \\ & x_4 + \frac{1}{2}x_5 + 6s = \frac{5}{2} \\ & x_1, x_2, x_3, x_4, x_5, s \geq 0, \end{cases}$$

avec $UB_{S1} = \frac{15}{2}$ et $\bar{x}^{(S1)} = (2, \frac{1}{2})$.

S_1 ne peut pas éliminer, alors la même règle de branche. On crée deux nouvelles nœuds :

$$S_{11} = S_1 \cup \{x; x_2 \leq 0\} \qquad \text{et} \qquad S_{12} = S_1 \cup \{x; x_2 \geq 1\}.$$

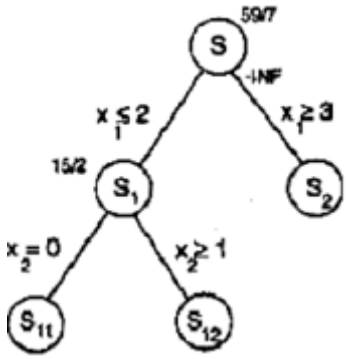


FIGURE 3.7 – Une partielle branche 2.

L'ensemble des nœuds active est S_2, S_{11}, S_{12} on choisis S_2 arbitrairement.

Pour résoudre $LP(S_2)$ on utilise l'algorithme dual simplexe la même méthode comme avant. La contrainte $x_1 > 3$, d'abord écrite $x_1 - t = 3, t \geq 0$, et

avec les variables d'écarts on obtient : $\frac{1}{7}x_3 + \frac{2}{7}x_4 + t = -\frac{1}{7}$

$$\begin{cases} UB_{S2} = & \max \frac{59}{7} - \frac{4}{7}x_3 - \frac{1}{7}x_4 \\ & x_1 + \frac{1}{7}x_3 + \frac{2}{7}x_4 = \frac{20}{7} \\ & x_2 + x_4 = 3 \\ & -\frac{2}{7}x_3 + \frac{10}{7}x_4 + x_5 = \frac{23}{7} \\ & \frac{1}{7}x_3 + \frac{2}{7}x_4 + t = -\frac{1}{7} \\ & x_1, x_2, x_3, x_4, x_5, s, t \geq 0 \end{cases}$$

On obtient : $UB_{S2} = -\infty$ (est impossible), donc S_2 est éliminé.

Pour l'instant, la liste de nœuds est S_{11}, S_{12}, on choisis S_{12} arbitrairement,

$$S_{12} = S \cap \{x : x_1 \leq 2, x_2 \geq 1.\}$$

Le programme linéaire résulté à pour solution optimale $\bar{x}^{S12} = (1, 2)$ avec valeur 7, comme $\bar{x}^{S12}$ est entier $UB^{S12} = 7$.

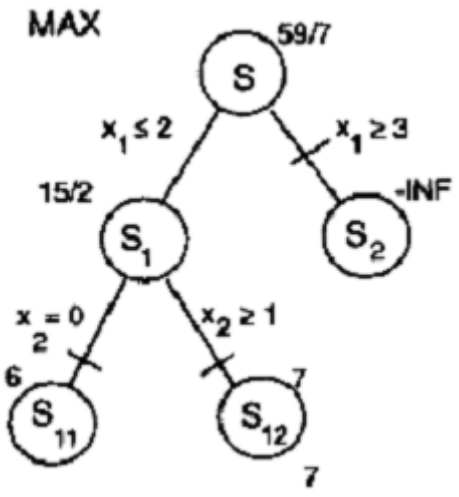


FIGURE 3.8 – Branche finale .

La solution de $LP(S_{12})$ est entier, on adapte le méllieur valeur travant alors : $7 = LB \leftarrow \max(LB, 7)$ et on mémorisé la solution corespondant $(2, 1)$ et maintenant on élimine S_{12} par optimalité.Il reste dans la listes des nœuds seulement

S_{11}, $S_{11} = S \cap \{x : x_1 \leq 2, x_2 \leq 0\}$. Le programme linéaire résulté a $\bar{x}^{S11} = (\frac{3}{2}, 0)$ pour solution optimale avec valeur 6. Comme $LB = 7 > UB^{S11} = 6$, le nœud est éliminé par borne.

Comme la liste des nœuds est vide, l'algorithme termine et donc la solution est $x = (2, 1)$ avec valeur optimale $Z^{(ILP)} = 7$.

Chapitre 4

Algorithme Branch and Bound de Little pour TSP

En 1963, J.D.C. Little, Lawler, E.L. et Wood, D.E [7]. ont présente une méthode rigoureuse d'optimisation pour le problème TSP. Cette méthode peut être appliquée pour les problèmes symétriques ainsi que les problèmes asymétriques.

Dans le problème du voyageur de commerce, on a donc un graphe connexe $G = (X, U)$ dans lequel chaque arc $u = (i, j)$ est muni d'une longueur $c(i, j)$ et ce graphe est d'ordre N ; où X est l'ensemble des sommets et U l'ensemble des arcs. L'objectif est de chercher un chemin minimal passant par tous les sommets une seule et unique fois sauf pour le sommet de départ sur lequel on revient. On cherche donc à minimiser la somme des distances parcourues entre les sommets d'une chaîne élémentaire.

Pour ce faire, on va utiliser l'algorithme développé par Little, cet algorithme permet de déterminer un circuit hamiltonien minimal.

4.1 Principe de L'algorithme de Little

L'algorithme de Little est un algorithme de résolution du TSP par la méthode Branch and Bound. Il utilise des opérations de réduction qui sont très similaires à ceux utilisés dans la méthode hongroise pour résoudre les problèmes d'affectation.

La direction générale de la méthode branch and bound de Little est basé sur deux principes :

1 Soit $z(t)$ le coût total d'un circuit t arbitraires lorsque C est la matice de coût. Supposons qu' on soustrait une constante e_i de chaque élément de la ligne i de C $(i = 1, ..., n)$ et puis soustraire une constante f_j de chaque élément dans la $j^{ème}$ colonne de la matrice résultante $(j = 1, ..., n)$. Ensuite, le coût total $\grave{z}(t)$ d'un circuit t arbitraire pour la matrice réduite $\grave{C} = (c_{ij} - e_i - f_j)$ est égal à :

$$\grave{z}(t) = z(t) - \sum_{i=1}^{n} e_i - \sum_{j=1}^{n} f_j, \tag{4.1}$$

ce qui signifie que le tour est optimal, pour la matrice des coûts initiale C, si et seulement si elle est optimale pour la matrice des coûts réduits $\grave{C}$ car $\sum_{i=1}^{n} e_i - \sum_{j=1}^{n} f_j$ est indépendant de t.

2 Soit $z(t)$ les coûts d'une tournée t arbitraires lorsque la matrice des coûts est C. Si cette tournée ne contient pas l'arc (a, b) alors

$$z(t) \geq \min_{x \neq b} c_{ax} + \min_{y \neq a} c_{yb}$$

Ce résultat découle directement si nous remarquons que si t ne contient pas l'arc (a, b), il doit donc contenir les arcs (a, x) et(y, b) avec $x \neq b$ et $y \neq a$.

Le nombre sur le côté droit de la l'inégalité ci-dessus, peut être considéré

comme des coûts de pénalité. Pour que l'arc (a, b) ne soit pas compris dans la tournée.

Nous allons maintenant formuler la direction générale et la méthode B&B de Little .

- La séparation consiste à considérer l'inclusion ou l'exclusion d'un trajet (i, j) dans une tournée. Chaque séparation produisant deux branches, l'arbre de recherche est binaire.

- L'évaluation fournit une inférieure du coût de la tournée en effectuant des opérations sur la matrice de coûts. De l'équation (4.1) nous obtenons une borne Inférieure LB à destination de nœud F composé de tous les circuits est donné par :

$$LB(F) = \sum_{i=1}^{n} e_i + \sum_{j=1}^{n} f_j = h_0$$

- On développe l'arbre de façon préférentielle vers la droite (inclusion de trajets) pour rapidement aboutir à une tournée complète T, de coût c.
- On développe ensuite les autres branches pour tenter de trouver une meilleure tournée que T. Dés qu'un nœud a un coup supérieur ou égal à c, on peut interrompre la branche.

Règle de branchement

Le nœud constitué de tous les tours est divisé en deux sous-ensembles de tournées en de la façon suivante.

On Calcule pour chaque arc (i, j) pour lequel $\grave{c}_{ij} = 0$ "coûts de pénalité " le $P'_{ij} = \underset{x \neq j}{min}\, c'_{ix} + \underset{y \neq i}{min}\, c'_{yj}$. soit (a, b) soit un arc (i, j) pour laquel P'_{ij} est maximal. On divisé l'ensemble des visites dans le sous-ensemble de circuits qui contiennent l'arc (a, b) et le sous-ensemble de circuits qui ne le pas. Pour le nœud constitué de tous les tours contenant l'arc (a, b), on modifié la matrice des coûts réduits $\grave{C}$ en supprimant la a^{eme} ligne et la b^{eme}colonne et en définissant l'élément $d'_{ba} = \infty$ pour éviter le sous-tours. La borne inférieure de ce nœud peut maintenant être améliorés par $h_0 + h_1$ en appliquant les opérations de réduction décrites pour la matrice de moindre coût. Pour le nœud constitué de tous les tours ne contenant pas d'arc (a, b), la matrice des coûts réduits $\grave{C}$ est modifié par la mise de $c'_{ab} = \infty$. La borne inférieure de ce nœud peut maintenant être amélioré par $h_0 + P'_{ab}$.

4.1.1 Exemple

Nous allons illustrer cette méthode sur un exemple de TSP symétriques avec 5 villes

	1	**2**	**3**	**4**	**5**
1	–	10	8	9	7
2	10	–	10	5	6
3	8	10	–	8	9
4	9	5	8	–	6
5	7	6	9	6	–

Si nous commençons à réduire les lignes puis les colonnes, nous obtenons la matrice suivante :

	1	**2**	**3**	**4**	**5**	e_i
1	$-$	3	**0**	2	0	7
2	5	$-$	4	0	1	5
3	0	2	$-$	0	1	8
4	4	0	2	$-$	1	5
5	1	0	2	0	$-$	6
f_j	0	0	1	0	0	32

Avec la borne inférieure est :

$$LB(F) = \sum_{i=1}^{5} e_i + \sum_{j}^{5} f_j = h_0 = 32$$

	1	**2**	**3**	**4**	**5**
1	$-$	3	$\mathbf{0^{(2)}}$	2	$0^{(1)}$
2	5	$-$	4	$0^{(1)}$	1
3	$0^{(1)}$	2	$-$	$0^{(0)}$	1
4	4	$0^{(1)}$	2	$-$	1
5	1	$0^{(0)}$	2	$0^{(0)}$	$-$

P'_{ij} est maximal pour $(i,j) = (1,3)$ c-a-d :$P'_{13} = 2$.

On divise par rapport à la variable x_{13} .

$x_{13} = 1$

	1	**2**	**4**	**5**
2	5	$-$	0	1
3	**0**	2	0	1
4	4	0	$-$	1
5	1	0	0	$-$

$\xrightarrow{(c'_{31} = \infty)}$

	1	**2**	**4**	**5**	e_i
2	4	$-$	0	0	0
3	$-$	2	0	0	0
4	3	0	$-$	0	0
5	0	0	0	$-$	0
f_j	1	0	0	1	$h_1 = 2$

$\rightarrow$

	1	**2**	**4**	**5**
2	4	–	$0^{(0)}$	$0^{(0)}$
3	–	2	$0^{(0)}$	$0^{(0)}$
4	3	$0^{(0)}$	–	$0^{(0)}$
5	$\mathbf{0^{(3)}}$	$0^{(0)}$	$0^{(0)}$	–

$LB(F_1) = h_0 + h_1 = 32 + 2 = 34$

On divise par rapport à x_{51} :

$x_{13} = x_{51} = 1$

	2	**4**	**5**
2	–	0	0
3	2	0	0
4	0	–	0

$\xrightarrow{(c'_{35} = \infty)}$

	2	**4**	**5**	e_i
2	–	0	0	0
3	2	**0**	–	0
4	0	–	0	0
f_j	0	0	0	$h_2 = 0$

$\rightarrow$

	2	**4**	**5**
2	–	$0^{(0)}$	$0^{(0)}$
3	2	$\mathbf{0^{(2)}}$	–
4	$0^{(2)}$	–	$0^{(0)}$

$LB(F_3) = h_1 + h_2 = 34 + 0 = 34$

On divise par rapport à x_{34} :

$x_{13} = x_{51} = x_{34} = 1$

	2	**5**
2	–	0
4	0	0

$\xrightarrow{(c'_{45} = \infty)}$

	2	**5**
2	–	0
4	0	–

$x_{13} = x_{51} = x_{34} = x_{25} = x_{42} = 1$.

Nous obtenons la tournée $(1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 1)$, $UB^{best} = 34$.

Pour le noeud $x_{13} = 0$ $(c_{13} = \infty)$ nous trouvons :

$$LB(F_2) = h_1 + P'_{13} = 32 + 2 = 34 \geq UB^{best}.$$

Pour le noeud $x_{51} = 0$ $(c_{51} = \infty)$ nous trouvons :

$$LB(F_4) = h_2 + P'_{51} = 34 + 3 = 37 \geq UB^{best}.$$

Pour le noeud $x_{32} = 0$ ($c_{32} = \infty$) nous trouvons :

$$LB(F_6) = h_3 + P'_{32} = 34 + 2 = 36 \geq UB^{best}$$

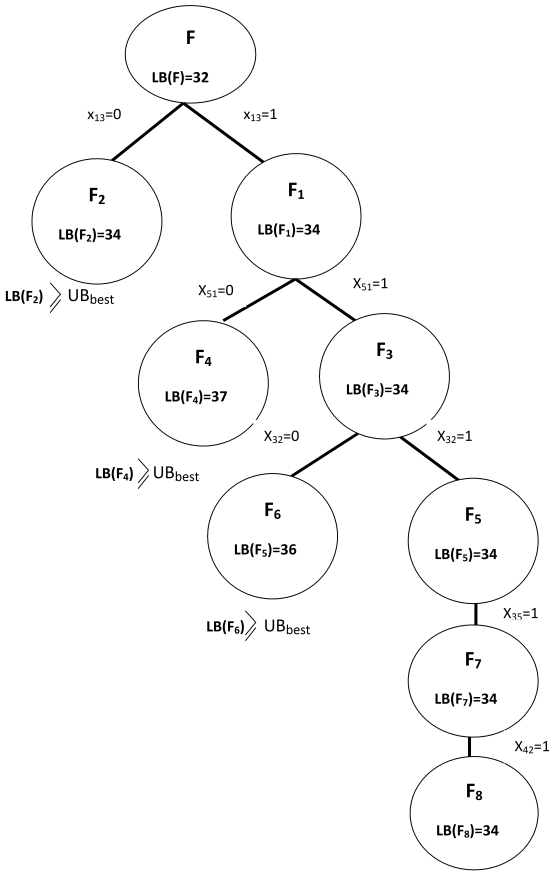


FIGURE 4.1 – Arbre représentatif de l'exemple 4.1.1

Sur MATLAB

L'exécution de code 1 se donne :

La tournée optimale est : $(1 \to 3 \to 4 \to 2 \to 5 \to 1)$

$UB^{best} = 34$.

CODE 1 :

```
clear all, clc
str=['++++++++ Matrice : contrainte de dégré +++++++++'];
disp(str)
A=[0 1 0 0 0 1 zeros(1,19);0 0 1 0 0 0 0 0 0 0 1 zeros(1,14);
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 zeros(1,9);
0 0 0 0 1 zeros(1,15) 1 0 0 0 0;
0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 zeros(1,10);
zeros(1,8) 1 0 0 0 0 0 0 0 1 zeros(1,8);
zeros(1,9) 1 zeros(1,11) 1 0 0 0;
zeros(1,13) 1 0 0 0 1 zeros(1,7);
zeros(1,14) 1 zeros(1,7) 1 0 0;
zeros(1,19) 1 0 0 0 1 0 ]
str=['++++ Matrice : contrainte d'Elimin de sous tours ++++'];
disp(str)
Aeq=[1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
0 0 0 0 0 0 0 0 0 0  1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0;
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0;
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 ;eye(5,5) eye(5,5)
eye(5,5) eye(5,5) eye(5,5) eye(5,5)]
str=['++++++++++ second membre +++++++++'];
disp(str)
```

```
beq = [1; 1; 1; 1;1;1;1;1;1;1]
str=['++++++++++ Matrice des couts ++++++++++'];
disp(str)
f = [999999;10; 8;9;7;10;99999999;10;5;6;8;10;
9999999;8;9;9;5;8;9999999;6;7;6;9;6;9999999999999]
str=['++++++++++ SOLUTION ++++++++++++'];
disp(str)
[x,fval,exitflag,output]= bintprog(f,A,beq,Aeq,beq)
str=['+++++++++++++++++ END +++++++++++++++++'];
disp(str)
 1-tree
f = [999999 10 8 9 7;10 99999999 10 5 6;8 10 9999999 8 9;
9 5 8 9999999 6;7 6 9 6 9999999999999]
ff=[x(1:5)';x(6:10)';x(11:15)';x(16:20)';x(21:25)']
fff=zeros(5,5);
for i=1:5
for j=1:5
if ff(i,j)==1
fff(i,j)=f(i,j);
end end end
fff
view(biograph(f,[],'ShowArrows','off','ShowWeights','on'))
view(biograph(ff,[],'ShowArrows','off','ShowWeights','on'))
```

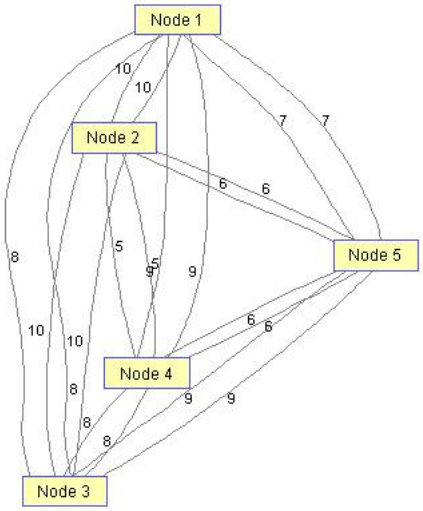


FIGURE 4.2 – Les tours possibles.

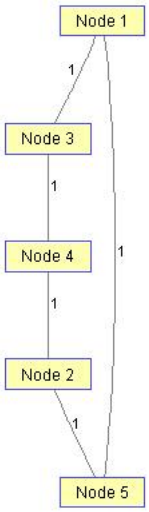


FIGURE 4.3 – Le tour optimale.

4.2 Algorithme de Little

Nous allons présenter simultanément l'algorithme et élaborer un exemple d'application.

L'algorithme de Little est du type *"séparation et évaluation"* : on effectue un parcours en profondeur de l'arborescence binaire des choix possibles, en attribuant à chaque noeud une évaluation par défaut, et une valeur de pénalité à chaque choix auquel on renonce. On remarque tout d'abord que l'on ne change pas le problème en soustrayant un nombre quelconque à une ligne ou une colonne de la matrice des distances entre les villes (cela revient à soustraire cette distance à tous les circuits hamiltoniens). On appellera réduction de la matrice l'opération consistant à soustraire de chaque ligne le plus petit élément, puis à soustraire le plus petit élément de chaque colonne de la matrice ainsi obtenue (en ignorant les éléments diagonaux, qu'on peut conventionnellement initialiser à ∞).

L'évaluation par défaut associée à la racine de l'arborescence est la somme des nombres soustraits lors de cette réduction.

Chaque noeud de l'arborescence correspond à un choix (fait ou abandonné).

L'évaluation par défaut associée à un noeud de type "*choix fait*" est la somme de l'évaluation par défaut associée au noeud père, et d'une évaluation par défaut du chemin restant à parcourir (obtenue par réduction de la matrice des liaisons encore possibles ; il faut prendre soin d'éliminer les liaisons créant des circuits parasites.

Le pénalité associé à un choix abandonné est calculé de la façon suivante : si on renonce à une liaison $s \to t$, il faudra sortir de s et entrer dans t par d'autres liaisons ; la valeur de pénalité associée à $s \to t$ est la somme du plus petit coût des liaisons $s \to s'$ et du plus petit coût des liaisons $t' \to t$ encore disponibles, avec $s \neq s'$ et $t \neq t'$.

L'évaluation par défaut associée à un noeud de type "*choix abandonné*" est la somme du pénalité et de l'évaluation par défaut du noeud père.

On parcourt l'arborescence en profondeur, en ignorant les noeuds dont l'évaluation par défaut dépasse la valeur d'un circuit hamiltonien déjà trouvé.

Notez qu'après une réduction, le graphe n'est plus symétrique. Les arcs entrant et sortant n'ont plus même valeur : le graphe est orienté.

Algorithme de Little :

maxi := $+\infty$

s (racine) :=ville de départ

calculer e (racine)

POUR toutes les liaisons $s(racine) \rightarrow v$ possibles

calculer le pénalité pour $s(racine) \rightarrow v$

FIN POUR

soit $s(racine) \rightarrow v$ une liaison de pénalité maximal P_{sv} ;

créer les fils v et $\overline{v}$ de racine ;

$s(v) := v$;

calculer $e(v)$;

visiter v ;

$s(\overline{v}) := \overline{v}$;

$e(\overline{v}) := e(racine) + P_{sv}$;

visiter $\overline{v}$;

visiter un noeud n :

SI $e(n) \leqslant maxi$ **ALORS**

Si $s(n) \neq ville\ de\ départ$ **ALORS**

POUR toutes les liaisons $s(n) \rightarrow v$ possibles

calculer le pénalité pour $s(n) \rightarrow v$

FIN POUR

$s(n) \rightarrow v$ une liaison de pénalité maximal P_{sv} ;

les fils v et $\overline{v}$ de n ;

$s(v) := v$;

calculer $e(v)$;

visiter v ;

$s(\overline{v}) := s(n)$;

$e(\overline{v}) := e(n) + P_{sv}$;

visiter $\overline{v}$;

SINON

$maxi := e(n)$

FIN SI

FIN SI

Choix fait :

On élimine les arcs sortants du sommet d'origine et les arcs entrant dans le sommet extrémité. On recalcule une réduction.

Choix abandonné :

On élimine l'arc en question, et on recalcule une réduction.

4.2.1 Résultats Numériques

L'algorithme de Little est codé en C est exécuté sur un PC disposant d'un processeur Intel cor i3 avec une fréquence de 2.4 GHZ et une mémoire vive de 4Go tournant sous Microsoft Windows 7 et il est testé sur quelques instances de TSPLIB ([4]).

Les instances sont donnés de problèmes réel, sont on connait la solution optimale . A titre d'exemple, Considérons la matrice du Tableau 1.

∞	27	43	16	30	26
7	∞	16	1	30	25
20	13	∞	35	5	0
21	16	25	∞	18	18
12	46	27	48	∞	5
23	5	5	9	5	∞

Tableau 1 : Matrice originale des coûts

L'entrée à la position (i, j) noté $c(i, j)$, représente le coût (distance) pour aller de la ville i à ville j.

l'execution du code 2 a donne les résultats suivants :

```
Le cas ou n'existe pas un cout entre i et j, on prend -1 :
-1    10   8   9   7
 10   -1   10  5   6
 8    10   -1  8   9
 9     5   8  -1   6
 7     6   9   6  -1
 Le tour optimale est :
     1  3
     5  1
     3  4
     2  5
     4  2
```

CODE 2 :

```
/* run this program using the console pauser or add your own getch,
```

```
/*
Nom: zahia ahmedi ezzourgui
Description:  programme de la méthode de little branch and bound
*/
#include <stdio.h> /* printf, scanf */
#include <stdlib.h> /*malloc*/
/*  l'nstruction #include <stdio.h> permet
 de utilise les mots clés de language c et
 elle est le plus grand bibliothèque */
/*---------------------------------------*/
/*je définie  un nouvelle type noté Node*/
typedef struct node* pNode;
/*je définie  un nouvelle type de pointeur noté pNode*/
typedef struct node{
double cout[100][100];
int paire[2];
double inf;
int suprimel[100];
int suprimec[100];
int iteration;
pNode left;
pNode right;
}Node;
/*---------------------------------------*/
/* cette fonction réduit le minimum d'une ligne i*/
double minl ( int n,double c[100][100],int i,
```

```
int* suprime_col,double* min )
{
/*j est l'indice de colonne*/
int j=0;
/* min[n] est le tableau des minimums des lignes  */
/* on parcoure la ligne i colonne par colonne jusqua le premiere
 donné differente à (-1) ou la colonne est suprimé  */
while((c[i][j]==-1)||(suprime_col[j]==1))
{
j=j+1;
}
/* le minimum de la ligne i min[i] garde la premier doonné*/
min[i]=c[i][j];
/*puis on parcour jusqua  le minimum*/
j=j+1;
while(j<n)
{
if((c[i][j]!=-1)&&(suprime_col[j]!=1)&&(c[i][j] < min[i]))
{
min[i]=c[i][j];

}
j=j+1;
}
return min[i];
}
/*-------------------------------------------*/
```

```
/*cette fonction réduit le minimum d'un colonne j */
double minc (int n,double c[100][100],int j
,int* suprime_ligne, double* miin)
{
int i=0;
/* i est lindice des ligne*/
while((c[i][j]==-1)||(suprime_ligne[i]==1))
{
/* le cas de ligne suprimé ou un case négatif*/
i=i+1;
}
miin[j]=c[i][j];
i=i+1;
while(i<n)
{
if((c[i][j]!=-1)&&(suprime_ligne[i]!=1)&&(c[i][j]< miin[j]))
{
miin[j]=c[i][j];
}
i=i+1;
}
return miin[j];
}
/*---------------------------------------*/
/* cette fonction réduit  la matrice des cout par ligne*/
void reduction_ligne(int n,double c[100][100]
,int* suprime_ligne,
```

```
int* suprime_col,double* min)
{
int i,j;
double oui;
/* i est l'indice des lignes et j est l'indice des colonnes*/
for(i=0;i<n;i++)
{
if (suprime_ligne[i]!=1)
{ oui=minl(n ,c ,i , suprime_col, min);
for(j=0;j<n;j++)
{
if ((suprime_col[j]!=1) && (c[i][j]!=-1))
{
c[i][j]=c[i][j]-oui;
}
}
}
}
}
/*-----------------------------*/
/* cette fonction réduit  la matrice des cout par colonne*/
void reduction_col(int n,double c[100][100],int* suprime_ligne
,int* suprime_col,double* miin)
{
int i,j;
double oui;
/* i est l'indice des lignes et j est l'indice des colonnes*/
```

```
for(j=0;j<n;j++)
{
if (suprime_col[j]!=1)
{ oui=minc(n,c,j,suprime_ligne,miin);
for(i=0;i<n;i++)
{
if ((suprime_ligne[i]!=1)&&(c[i][j]!=-1))
{
c[i][j]=c[i][j]-oui;
}
}
}
}
}
/*--------------------------------------*/
/* cette fonction réduit  la  somme des réduites de la matrice des
 cout par colonne et ligne*/
double sume( int n,double c[100][100],int* suprime_ligne
,int* suprime_col,double* min,double* miin)
{
int i;
double sum = 0;
for(i=0;i<n;i++)
{
if ((suprime_ligne[i]!=1) && (suprime_col[i]!=1))
{
sum =sum + min[i]+miin[i];
```

```
}
else if((suprime_ligne[i]!=1) && (suprime_col[i]==1))
{
sum =sum+ min[i];
}
else if((suprime_ligne[i]==1) && (suprime_col[i]!=1))
{
sum =sum+ miin[i];
}
}
/*-------------------------------------*/
/*sum sera la somme des minimum des lignes plus les minimum
des colonnes non suprimé*/
return sum;
/*sum sera la borne inférieur au debut*/
}
/*-------------------------------------*/
/* cette fonction réduit le 2 éme minimum de la(les) colonne(s) l */
double min_col2 ( int n , int i , int* zero , double c[100][100] ,
 double* colon , int* fin , int* suprime_ligne )
{
/*i est l'indice de la ligne*/
/*-----------*/
int l ,j , a , k ;
double minc2=0.0;
double min_colon[100], max ;
/*-------------------------*/
```

```
for(k=0;k<fin[i];k++)
{
/* le vecteur fin contenant le dernier colonne null dans
la ligne i */
/* zero est la matrice qui contenant les colonnes
 null dans
  la ligne i aprés la réduction */
/* l sera contenant la colonne de 0  */
l=zero[i * n + k];
/*  j sera contenant l'indice des lignes  */
j=0;
/* je cherche le premier élément de la matrice c disponible
 telle que */
while(((c[j][l]==-1)||(j==i)||(suprime_ligne[j]==1)) && (j<n))
{
j=j+1;
/*sauter la ligne actuale*/
}
/*on enregitre en minc2 le premier terme aprés les testes*/
minc2=c[j][l];
a=j+1;
for(j=a;j<n;j++)
{
if((c[j][l]!=-1)&&(j!=i)&&(suprime_ligne[j]!=1)&&(c[j][l] < minc2))
{
minc2 = c[j][l];
}
```

```
}
min_colon[k] = minc2 ;
if (k == 0)
{
max=min_colon[0] ;
*colon=(double)l ;
}
/* s'il existe plus d'un 0 dans la ligne i choisir
 le maximum des min2*/
else if ( max <= min_colon[k] )
{
max = min_colon[k] ;
/*colon sera contenant la colonne de 0 le plus grand min2
 pour colonne*/
*colon= (double)l;
}
}
return max;
}
/*--------------------------------*/
/*cette fonction réduit le 2 éme minimum de la ligne(s) l*/
double min_ligne2( int n,int i, int* zero, double c[100][100],
int* suprime_col )
{
int l,j,a;
double minl2=0.0;
/* l sera contenant la colonne de 0  */
```

```
l=zero[i * n + 0];
/*  j sera le conteur d'indice des colonne */
j=0;
/* je cherche le premier élément de la matrice c disponible */
while(((c[i][j]==-1)||(j==l)||(suprime_col[j]==1))&&(j<n))
{
j=j+1;
}
/*on enregitre en minl2*/
minl2=c[i][j] ;
a=j+1;
for(j=a;j<n;j++)
{
if((c[i][j]!=-1)&&(j!=l)&&(suprime_col[j]!=1)&&(c[i][j]<minl2))
{
minl2 = c[i][j];
}
}

return minl2;
}

/*----------------------------------------*/
/* cette fonction réduit  les indice des zero de la matrice
 des cout réduite  par colonne et ligne*/

void recherche_zero(int n ,int* zero,double c[100][100]
```

```
,double puissance[100][2],double colon
,int* suprime_ligne,int* suprime_col,
int* fin)
{
int k=0;
int i,j;
/*i est l'indice de  ligne et j est l'indice de colonne*/
/* on cherche d'abord  les 0 en chaque ligne  et en garde
 en la matrice zero et ces nombres en  le vecteur fin */
/*  ie: fin sera contient  le nombre des zéro en chaque ligne i */
for(i=0;i<n;i++)
{
if  (suprime_ligne[i]!=1)
{
k=0;
for(j=0;j<n;j++)
{
if ((c[i][j]==0)&&(suprime_col[j]!=1))
{
zero[i * n + k]=j ;
/*zero sera contient le colonne  (de 0) de la ligne i*/
/* il existe une probabilité que la ligne sera contenant plus de 0*/
k=k+1 ;
}

}
/* fin[i] est le nombre de 0 en la ligne i*/
```

```
fin[i]=k;
/* remplire les puissances de chaque ligne */
if(fin[i]==1)
{
/*s'il existe un seul zero en la ligne i */
puissance[i][0]=min_ligne2 (n,i,zero,c,suprime_col ) +
min_col2 (n,i,zero,c,&colon,fin,suprime_ligne) ;
puissance[i][1]=(double)zero[i * n + 0] ;
}
else
{
puissance[i][0]=min_col2 (n,i, zero,c,&colon,fin,suprime_ligne) ;
puissance[i][1]=colon ;

}
}
}
}
/*------------------------------*/
/* cette  fonction  permet de donner le maximum de
 la matrice  puissanse */
double max_puis( int n , double puissance[100][2] , int* ligne ,
int* colonne , int* suprime_ligne)
{
double max=0;
int i;
for(i=0;i<n;i++)
```

```
{
if (( suprime_ligne[i]!=1) && (puissance[i][0] > max ))
{
max=puissance[i][0];
*ligne=i;

*colonne=(int) puissance[i][1];
}
}
return max;
}
/*---------------------------*/
/* la fonction principale " void main"*/
void main()
{
int n;
printf("Donnez la taille de la matrice des couts: ");
scanf("%d",&n);
printf("le cas ou n'existe pas un cout entre i et j ,on prend -1\n");
int i,j;
double c[100][100];
printf("pour passe au autre colonne de la matrice
  de meme ligne faire 'espace'\n pour
   aller au ligne suivant faire 'entrer'\n");
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
```

```
{
scanf("%lf",&c[i][j]);
}
}
double* min;
min=(double*)malloc( n*sizeof(double));
double* miin;
miin=(double*)malloc( n*sizeof(double));
int ligne=0;
int colonne=0;
double colon;
int k=0;
double borne=0;
double haut=0;
int iter=0;
pNode n1,p;
pNode point[1000];
int* suprime_ligne;
suprime_ligne=(int*)malloc( n*sizeof(int));
int* suprime_col;
suprime_col=(int*)malloc( n*sizeof(int));
double borne_inf;
int nbre_stop;
int* fin;
fin=(int*)malloc( n*sizeof(int));
int* zero;
zero=(int*)malloc( n*n*sizeof(int));
```

```
for(i=0;i<n;i++)
{
for( j=0; j<n; j++ )
{
/*initialisation de la matrice dynamique */
zero[i * n +j] = 0 ;
}
}
double puissance[100][2];
double minimum=0;
int finale=0;
int premier;
/* tour est la matrice des paire de ville qui peut etre un tour
 optimale */
int tour[100][2];
int optimale[100][2];
/* on intialise les tableau suprime de ligne et de colonne*/
for(i=0;i<n;i++)
{
suprime_ligne[i]=0;
suprime_col[i]=0;
}
/* on crée un pointeur vers un Node ie: elle contient s'adresse */
n1=(pNode)malloc(sizeof(Node));
premier=1;
for(i=0 ; i<n ; i++)
{
```

```
n1->suprimel[i] = suprime_ligne[i] ;
n1->suprimec[i] = suprime_col[i]  ;
}
while(finale==0)
{
while (iter<n-1)
{
/* on va  réduit les lignes et les colonne de la matrice de n1 */
reduction_ligne ( n , c , suprime_ligne , suprime_col , min ) ;
reduction_col ( n , c , suprime_ligne , suprime_col , miin ) ;
/* puis on cherche les zero de la matrice */
recherche_zero( n , zero , c , puissance , colon , suprime_ligne
 , suprime_col , fin );
/* maintenant on remplire le premier noeud (père) */
for(i=0 ; i<n ; i++)
{
if(suprime_ligne[i]==0)
{
for(j=0;j<n;j++)
{
if(suprime_col[j]==0 )
{
n1->cout[i][j]=c[i][j];
}
}
}
}
```

```
borne=sume(n, c ,suprime_ligne, suprime_col,min,miin)+borne;
n1->inf=borne;
if ((premier!=1)&&( borne>minimum))
{
finale=1;
break;
}
printf("-------------------------------------") ;
printf("%lf\n",borne);
haut = max_puis( n , puissance , &ligne , &colonne , suprime_ligne) ;
n1->paire[0] = ligne ;
n1->paire[1] = colonne ;
printf("%d \t %d\n",n1->paire[0],n1->paire[1]) ;
c[colonne][ligne]= -1 ;
n1->cout[colonne][ligne]= -1 ;
suprime_ligne[ligne] = 1 ;
suprime_col[colonne] = 1 ;
n1->suprimel[ligne] = 1 ;
n1->suprimec[colonne] = 1 ;
n1->iteration=iter;
tour[iter][0]= ligne ;
tour[iter][1]=colonne;
if((iter!=0)&&(tour[iter-1][0]==tour[iter][1]))
{
n1->cout[tour[iter-1][1]][tour[iter][0]]=-1;
c[tour[iter-1][1]][tour[iter][0]]=-1;
}
```

```
for(i=0;i<n;i++)
{
if(suprime_ligne[i]==0)
{
for(j=0;j<n;j++)
{
if(suprime_col[j]==0)
printf("%d\t",(int) c[i][j]);
}
}
printf("\n");
}
/*on crée 2 noeud la droite d'abord*/
/*et on enregistre  l'adresse dans le champs right*/
n1->right=(pNode)malloc(sizeof(Node));
/* et on remplire cette noeud */
p=n1->right;
/*on crée un tableau de pointeur (adresse) des noeud droite*/
point[k]=p;
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if((i==ligne)&&(j==colonne))
p->cout[i][j]=-1;
else
p->cout[i][j]=c[i][j];
```

```
}
}
p->paire[0]=ligne;
p->paire[1]=colonne;
p->inf=borne+haut;
p->iteration=iter;
for(i=0 ; i<n ; i++)
{
p->suprimel[i] = suprime_ligne[i] ;
p->suprimec[i] = suprime_col[i]  ;
}
p->suprimel[ligne] = 0 ;
p->suprimec[colonne] = 0  ;
n1->left=(pNode)malloc(sizeof(Node));
n1=n1->left;
k=k+1;
iter=iter+1;
}
iter=iter-1;
reduction_ligne ( n , c , suprime_ligne , suprime_col , min ) ;
reduction_col ( n , c , suprime_ligne , suprime_col , miin ) ;
for(i=0;i<n;i++)
{
if(suprime_ligne[i]==0)
{
for(j=0;j<n;j++)
{
```

```
if((suprime_col[j]==0)&&(c[i][j]==0)&&(suprime_ligne[i]==0))
{
tour[iter][0]=i;
tour[iter][1]=j;
suprime_col[j]=1;
suprime_ligne[i]=1;
}
}
iter=iter+1;
reduction_ligne ( n , c , suprime_ligne , suprime_col , min ) ;
reduction_col ( n , c , suprime_ligne , suprime_col , miin ) ;
}
}
borne_inf=borne;
nbre_stop=k;
minimum=borne;
/* on cherche maintenant le plus petite borne et on le retour
à leur noeud */
for(k=0;k<nbre_stop;k++)
{
if((point[k]!=NULL)&&(point[k]->inf < borne_inf))
{
/* n1 resoit un pointeur vers  le minimmum des couts */
n1=point[k];
borne_inf=point[k]->inf ;
}
else if(point[k]->inf > borne_inf)
```

```
{
point[k]=NULL;
/*NULL : aucun adresse*/
}
}
/* il faut conserve le tour qu'il peut etre optimale
dans un vecteur appelé 'optimale'*/
if(finale==0)
{
for(i=0;i<n;i++)
{
optimale[i][0]=tour[i][0];
optimale[i][1]=tour[i][1];
suprime_col[i]=n1->suprimec[i];
suprime_ligne[i]=n1->suprimel[i];
for(j=0;j<n;j++)
{
c[i][j]=n1->cout[i][j];
}
}
}
iter=n1->iteration;
premier=2;
k=0;
}
printf("Le tour optimale est:\n");
for(k=0;k<n;k++)
```

```
{
printf(" %d   %d \n",optimale[k][0]+1,optimale[k][1]+1);
}
}
```

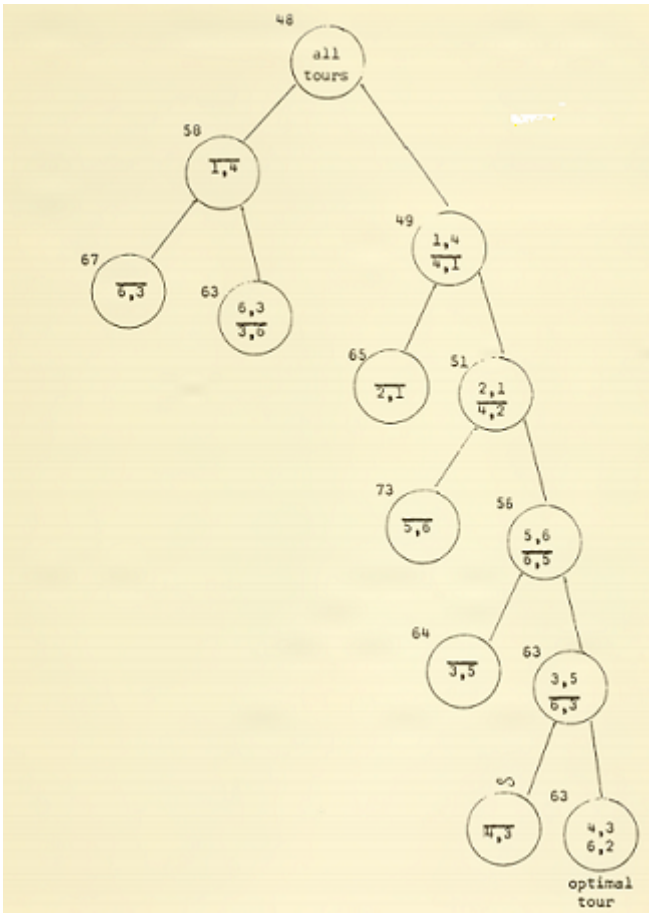


FIGURE 4.4 – Résolution de l'exemple par l'algorithme Branch and Bound de Little.

Conclusion

La méthode de séparation et évaluation est : notée **B & B** : est une des méthode permettant la résolution exacte de problèmes d'optimisation, notamment les problèmes d'optimisation combinatoires dont on cherche à minimiser le coût de la recherche. La méthode **B & B** propose un mécanisme de recherche très intelligent, grâce à lequel elle permet une bonne exploitation de l'espace de recherche et l'aboutissement à la solution optimale plus rapidement que d'autres méthodes exactes en combinant deux principes primordiaux : la séparation et l'évaluation.

Le point fort de cette méthode réside dans le fait qu'elle ne parcourt pas les sous branches dont on peut savoir à priori qu'elles ne permettent pas d'améliorer la solution rencontrée, ce qui est établi grâce aux bornes des noeuds. Cela permet de trouver de bonnes solution en un temps de recherche relativement court.

L'efficacité de cette méthode a attiré l'attention de nombreux chercheurs. Par conséquent, plusieurs améliorations de l'algorithme **B & B** ont été proposées, y compris les algorithmes : Branch and Cut (noté **B & C**) [10], Branch and Price (noté **B & P**) [2], Branch and Cut and Price (**B & C & P**).

Bibliographie

[1] Christine Solnon : Théorie des graphes et optimisation dans les graphes .

[2] Cynthia Barnhart , Ellis L.Johnson , George L.Nemhauser , Martin W.P.Savelsbergh , Pamela H.Vance : Branch-and-Price : Column Generation for Solving Huge Integer Problems. *Operations Research*, Vol , 46,No.3.(May-Jun,1998),pp.316-329.

[3] E.L.Lawler, J.K.Lenstra, A.H.G.Rinnooy kan, D.B.Shmoys D.B(Eds) : The travelling sales- man problem :A Guided Tour of Combinatorial Optimization. Wiley, 1985.

[4] G. Reinelt, TSPLIB : The travelling salseman problem Lirary,URL :http ://elib.zib.de/pub/mp-testdata/tsp/tsplib.html.

[5] H.H.Kuhn,the Hungarian Methode for the Assignment Problem,Naval Research Logistics Quarterly 2 (1955) 83-97.

[6] J. Teghem, Programmation linéaire, Edition Statistique et Mathématiques Appliquées (2003).

[7] J.D.C. Little, Lawler, E.L. et Wood, D.E, AN ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM, March 1, 1963

[8] L. LAMPORT, *LaTeX : A Document preparation system, Addison-Wesley, 1994*

[9] Métaheurstique, URL : http :// fr.wikipedia.org/wiki/métaheuristique.

[10] M.Padberg, G.Rinaldi : A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems. 1991.

[11] R.S.Garfinkel, G.L.Nemhauser : integer programing. Wiley(1972).