



Université Cheikh Anta Diop (UCAD)



Faculté Sciences et Techniques (FST)

Départements Mathématiques et Informatiques (DMI)

Mémoire Master II

Option : Mathématiques de la décision et Recherche opérationnelle

Titre :

Les Métaheuristiques en Optimisation

Par :

NGOM El hadji¹

Jury :

Président - **Prof.**

Membres **Prof.**
Prof.
Prof.

Superviseurs - **Prof. Salimata Guèye Diagne (UCAD)**
- **Prof. Youssou GNINGUE Université Laurentiane (Canada)**

Soutenue le

1. UCAD-AIMS-SENGAL

Dédicaces

Remerciements

Résumé

Abstract

Table des matières

Dédicaces	i
Remerciements	ii
Résumé	iii
Abstract	iv
Introduction générale	1
1 Les méthodes de résolution exactes et heuristiques	2
1.1 Introduction	2
1.2 Notions de complexité	2
1.3 Les méthodes exactes	3
1.3.1 La méthode du Simplexe	3
1.3.2 La Méthode de Séparation et d'Evauation	4
1.3.3 Methode des coupes planes ou cutting planes method	5
1.3.4 La méthode de Branch & Cut	6
1.4 Les heuristiques	11
1.4.1 La descente recursive (recherche locale)	11
1.5 Conclusion	11
2 Les métaheuristiques	12
2.1 Introduction	12
Définitions	12
Classification	12
2.2 Méthodes à base d'une seule solution	12
2.2.1 Le Recuit simulé	12
2.2.2 La recherche Tabou	12
2.3 Methodes à base de populations de solutions	12
2.3.1 Les colonies des fourmis	12
2.3.2 Les algo-génétiques	12
2.3.3 L'Optimisation par essaim de particule	12
2.3.4 La recherche dispersée	12
2.4 Conclusion	12

3 Exemples pratiques d'applications	13
3.1 Introduction	13
3.2 Example 1	13
3.3 Example 2	13
3.4 Example 3	13
3.5 Conclusion	13
Conclusion	14
References	15
Bibliographie	15

Table des figures

1.1	Arbre engendré par décomposition d'un problème	5
1.2	Structure de la methode de Gomory, (cf support de cours : PE, p.164) . . .	7
1.3	Leture du PL au format “.lp” par Cplex.	9
1.4	Details de résolution et affichage du résultat.	10

Liste des tableaux

Introduction générale

L'optimisation est un concept bien naturel dans la vie courante : devant un problème en présence de plusieurs solutions possibles, tout individu choisit (en général) une solution qualifiée de “meilleur”.

Les méthodes exactes des problèmes d'optimisation (combinatoire) permettent d'obtenir une solution dont l'optimalité est garantie ou assurée, dans certaines situations, on peut cependant chercher des solutions de bonne qualité, sans garantie d'optimalité mais avec un temps beaucoup considérablement réduit. Pour cela, on applique des méthodes appelées **métaheuristiques** (ce mot vient du grec *meta* qui signifie littéralement "au delà" mais '*à un plus haut niveau*' dans ce contexte), adaptées à chaque problème traité, avec seulement l'inconvénient de ne pas disposer en retour d'aucune information sur la qualité de la solution obtenue. Les heuristiques comme les métaheuristiques exploitent généralement des processus aléatoires dans l'exploration de l'espace de recherche pour faire face à l'explosion combinatoire engendrée par l'utilisation des méthodes exactes. Reposant sur une base stochastique, les métaheuristiques sont plus souvent itératives, ainsi le même processus de recherche est répété lors de la résolution. Les métaheuristiques acquièrent une grande avantage par leur capacité d'éviter les minima locaux en admettant des dégradations de la fonction objective au cours de la progression.

Grace à l'optimisation linéaire en nombres entiers, la discipline occupe une place capitale en mathématique décisionnaire, en recherche opérationnelle et en informatique. Des nombreuses applications peuvent être modélisées sous la forme de problème d'optimisation en nombres entiers telles que le voyageur de commerce, le problème de coloration de graphe, les problèmes de localisation etc. En général, on appelle programmation mathématique la recherche de l'optimum d'une fonction de plusieurs variables liées entre elles par des contraintes (sous formes'égalité ou d'inégalité). En particulier voici la forme standard d'une programmation linéaire :

$$\left\{ \begin{array}{l} \max Z(x) \\ s.c. \\ H_i(x) \leq \alpha_i \\ x \in S \subset \mathbb{R}^n \end{array} \right. \quad \left\{ \begin{array}{l} \min Z(x) \\ s.c. \\ H_i(x) \geq \alpha_i \\ x \in S \subset \mathbb{R}^n \end{array} \right.$$
$$i \in \{1, 2, \dots, m\}$$

Ici le vecteur $x \in \mathbb{Z}^n$ a pour composantes x_1, x_2, \dots, x_m qui les inconnues du problème. La fonction Z est la fonction objectif (on dit parfois aussi : fonction économique) et l'ensemble des conditions : $H_i(x) \leq \alpha_i$ ou $H_i(x) \geq \alpha_i$ avec $i \in \{1, 2, \dots, m\}$ et $x \in S$ sont les contraintes du problème.

LES MÉTHODES DE RÉOLUTION EXACTES ET HEURISTIQUES

1.1 Introduction

Résoudre un problème d'optimisation linéaire en nombres entiers consiste à chercher la meilleure solution possible pour ce problème, définie comme la solution globalement optimale ou un optimum global. La résolution de ces problèmes dits également combinatoires est souvent assez délicate puisque le nombre fini de solutions réalisables croît considérablement avec la taille du problème, ainsi que sa complexité. Cette remarque a poussé les chercheurs à développer de nombreuses méthodes de résolution aussi bien qu'en Recherche opérationnelle (RO) qu'en intelligence artificielle (IA).

Dans la section suivante, nous allons d'abord définir les notions de complexité quant aux problèmes d'optimisation combinatoire.

1.2 Notions de complexité

Avant de passer à un bref rappel sur les différentes méthodes de résolution exactes et heuristiques des problèmes d'optimisation linéaires en nombres entiers, nous introduisons quelques définitions et notions sur la complexité des ces derniers à savoir les PLNE.

La théorie de la complexité des algorithmes, née à la suite des travaux d'Emonds puis Cook et Karp, a justement pour objet de lier le nombre de calculs effectués lors de la résolution d'un problème au moyen d'un algorithme donné à la taille des données de ce problème. Nous ne ferons pas différence entre algorithme et programme.

En général, le temps d'exécution est le facteur majeur qui détermine l'efficacité d'un algorithme, alors la complexité en temps d'un algorithme est le nombre d'instructions nécessaires (affectation, comparaison, opérations algébriques, lecture et écriture, etc) que comprend cet algorithme pour une résolution d'un problème quelconque.

Définition 1.2.1. Une fonction $f(n)$ est $O(g(n))$ ou ($f(n)$ est de complexité $g(n)$) s'il existe un réel $c > 0$ et un entier positif n_0 tels que pour tout $n > n_0$ on a $|f(n)| \leq c.g(n)$.

Définition 1.2.2. 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 la taille de l'instance (ou sa longueur d'entrée).

Si k est le degré de ce polynôme en n , le problème correspondant est dit être résoluble en $O(n^k)$ et appartient à la classe P . La connexité d'un graphe est un exemple de problème polynomial de classe P .

Définition 1.2.3. La classe **NP** contient l'ensemble des 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 en temps 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 des solutions possibles et les tester à l'aide d'un algorithme polynomial.

Définition 1.2.4. On dit qu'un problème de recherche P_1 se réduit polynomialement à un problème de recherche P_2 par réduction de Turing s'il existe un algorithme A_1 pour résoudre P_1 en utilisant comme sous programme un algorithme A_2 résolvant P_2 , de telle sorte que la complexité A_1 est polynomiale, quand on évalue chaque appel de A_2 par une constante.

Définition 1.2.5 (La classe **NP-Complet**). Parmi l'ensemble des problèmes appartenant à **NP**, il existe un sous ensemble qui contient les problèmes les plus difficiles : on les appelle les problèmes **NP-Complets**. Un problème de classe **NP-Complet** possède la propriété que tous les problèmes **NP** lui sont réductibles. Si on trouve un algorithme polynomial à un problème **NP-Complet**, on trouve alors automatiquement une résolution polynomiale de tous les problèmes **NP** de la classe **NP**.

Définition 1.2.6 (La classe **NP-difficile**). Un problème est **NP-difficile** s'il est plus difficile qu'un problème **NP-Complet**, c'est-à-dire s'il existe un problème **NP-Complet** se réduisant à ce problème par une réduction de Turing.

Nous allons passer à la description des principales méthodes de résolution exactes des problèmes d'optimisation combinatoire.

1.3 Les méthodes exactes

Nous allons présenter d'abord quelques méthodes de la classe des algorithmes complets ou exactes, ces méthodes donnent une garantie de trouver la solution optimale pour une instance de taille finie dans un temps limité et de prouver son optimalité [6].

1.3.1 La méthode du Simplexe

Parmi ces méthodes, on peut remarquer l'algorithme du simplexe qui permet d'obtenir la solution optimale d'un problème d'optimisation en parcourant la fermeture convexe de l'ensemble de recherche. Son avantage c'est qu'il permet de résoudre un grand nombre de problèmes rapidement.

Voici la définition donnée par **Wikipédia**, (consulté le 25/12/2015) :

“L'algorithme du simplexe est un algorithme de résolution des problèmes d'optimisation linéaire. Il a été introduit par George Dantzig à partir de 1947. C'est probablement le premier algorithme permettant de minimiser une fonction sur un ensemble défini par des inégalités. De ce fait, il a beaucoup contribué au démarrage de l'optimisation numérique. L'algorithme du simplexe a longtemps été la méthode la plus utilisée pour résoudre les problèmes d'optimisation linéaire. Depuis les années 1985-90, il est concurrencé par les méthodes de points intérieurs, mais garde une place de choix dans certaines circonstances (en particulier si l'on a une idée des contraintes d'inégalité actives en la solution)”.

Cependant, l'algorithme du simplexe qui est bien entendu valable pour tout problème, il n'est nécessairement le plus efficace pour traiter des problèmes dont l'ensemble des contraintes présente certaines particularités (comme l'intégrité de la solution).

1.3.2 La Méthode de Séparation et d'Évaluation

L'algorithme de séparation et d'évaluation, connu sous l'appellation anglaise **Branch and Bound (B&B)**, est l'une des méthodes les plus connues pour résoudre des problèmes d'optimisation combinatoires NP-Complets comme le **problème de sac à dos**¹. Elle repose sur une méthode arborescente de recherche d'une solution optimale par séparations et évaluations, représentant les états solutions par un arbre d'états, avec des nœuds, et des feuilles.

Le branch and bound est basé sur trois axes principaux :

- ✎ l'évaluation,
- ✎ la séparation,
- ✎ la stratégie de parcours.

✎ L'évaluation :

L'évaluation permet de réduire l'espace en éliminant quelques sous-ensembles qui ne contiennent pas la solution optimale. L'objectif est d'essayer d'évaluer l'intérêt de l'exploration d'un sous-ensemble de l'arborescence. Le branch and bound utilise une élimination de branches dans l'arborescence de recherche de la manière suivante : la recherche d'une solution de coût minimal, consiste à mémoriser la solution la moins coûteuse rencontrée pendant l'exploration, et à comparer le coût de chaque nœud parcouru à celui de la meilleure solution. Si le coût du nœud considéré est supérieur au meilleur coût, on arrête l'exploration de la branche et toutes les solutions de cette branche seront nécessairement de coût plus élevé que la solution déjà trouvée.

✎ La séparation :

la séparation consiste à diviser le problème en sous-problèmes et en gardant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Cela revient à construire un arbre permettant d'énumérer toutes les solutions. L'ensemble des nœuds de l'arbre qu'il reste encore parcourir comme étant susceptibles de contenir une solution optimale, c'est-à-dire encore à diviser, est appelé ensemble des nœuds actifs. La procédure de séparation s'arrête lorsqu'une des conditions suivantes est vérifiée :

- ✓ on reconnaît la meilleure solution de l'ensemble,
- ✓ on reconnaît une solution meilleure que toutes celles de l'ensemble,
- ✓ on sait que l'ensemble ne contient aucune solution admissible.

✎ La stratégie de parcours :

- ✎ **La largeur d'abord** : cette stratégie favorise les sommets les plus proches de la racine en faisant moins de séparations du problème initial. Elle est moins efficace que les deux qui suivent,
- ✎ **La profondeur d'abord** : cette stratégie avantage les sommets les plus éloignés de la racine (de profondeur plus élevée) en appliquant plus de séparations au problème initial. Cette voie mène rapidement à une solution optimale en économisant la mémoire,
- ✎ **La meilleure d'abord** : cette stratégie consiste à explorer les sous-problèmes possédant la meilleure borne. Elle permet aussi d'éviter l'exploration de tous les sous-problèmes qui possèdent une mauvaise évaluation par rapport à la valeur optimale. Elle dirige également la recherche là où la probabilité de trouver une meilleure solution est plus forte.

1. Knapsack Problem (KP) ou problème de sac à dos est un problème simple et classique d'optimisation combinatoire appartenant à la classe NP-Complet.

Représentation graphique de la décomposition par B&B appliquée au KP :
Formulation mathématique du KP : Dans ce cas, nous avons un sac à dos de maximal P et n objets. Pour chaque objet i , nous avons le poids p_i et une valeur ou utilité c_i .
variables de décision :

$$x_i = \begin{cases} 1 & \text{si l'objet } i \text{ est mis dans le sac,} \\ 0 & \text{si l'objet } i \text{ n'est pas mis dans le sac.} \end{cases}$$

On obtient modèle suivant :

$$(KP) \begin{cases} \max \sum_{i=1}^n c_i x_i \\ s.c. \\ \sum_{i=1}^n p_i x_i \leq P \\ x_i \in \{0, 1\} \end{cases}$$

La recherche par décomposition de l'ensemble des solutions peut être représentée graphiquement par un arbre (voir figure 1.1).

C'est de cette représentation que vient le nom de "méthode de recherche arborescente".

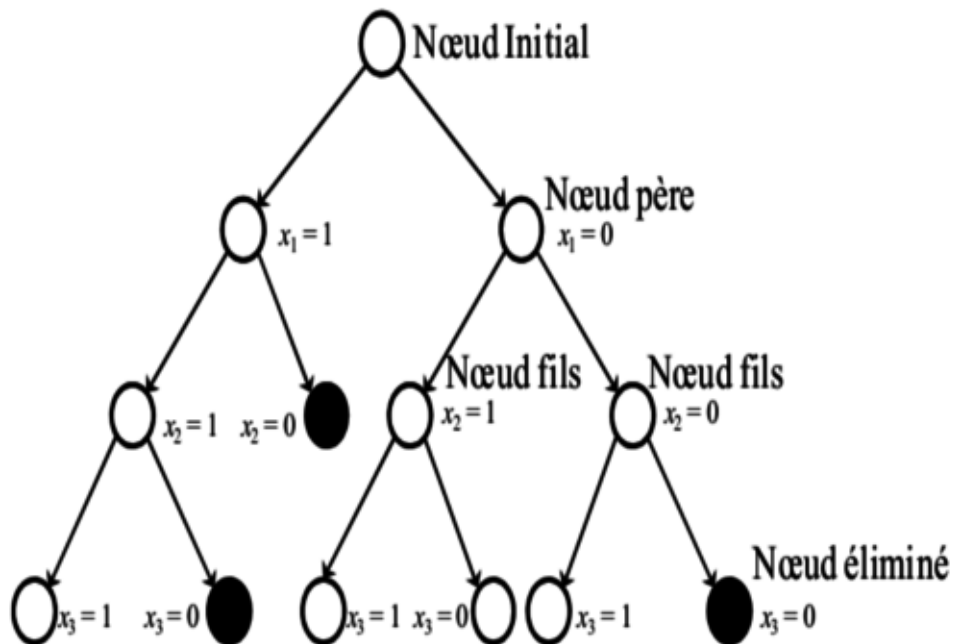


FIGURE 1.1 – Arbre engendré par décomposition d'un problème

1.3.3 Méthode des coupes planes ou cutting planes method

La méthode des coupes a été développée par A. Schrijver dans son livre intitulé *Theory of linear and integer programming* en 1986 mais les travaux de Ralph Edward Gomory ont rendu les coupes plus connues et plus efficaces. Elle est destinée à résoudre des problèmes d'optimisation linéaires en nombres entiers qui se formulent sous la forme standard d'un

programme linéaire (PL) :

$$(PL) \begin{cases} \min C^T x \\ s.c. \\ Ax \geq b \\ x \in \mathbb{R}^n \end{cases} \quad (1.1)$$

L'algorithme de cutting-plane forme une des classes des Algorithmes pour résoudre des PLE's qui utilise une idée sophistiquée très intéressante. A chaque étape, on diminue la région réalisable en utilisant un plan coupant (cutting plane) jusqu'à ce que la solution optimale du PL (**relaxation** du PLE) soit entière correspondant la solution optimale du PLE initial. **Idée de base :**

On ajoute des contraintes linéaires au PLE qui n'excluent pas la solution entière réalisable.

Stratégie :

1. On ajoute des contraintes linéaires au PLE et donc aussi à la relaxation, une à chaque étape, jusqu'à la solution optimale entière de la relaxation.
2. Puis qu'aucune solution réalisable du PLE n'est perdue par les coupes, alors la solution optimale entière de la relaxation du PLE ayant des contraintes ajoutées correspondra à la solution du PLE d'origine.

Observation :

Considérons un programme en nombres entiers sous la forme standard

$$PLE(I) \begin{cases} \min c^T x \\ s.c. \\ Ax = b \\ x \in \mathbb{Z}_+^n \end{cases} \quad (1.2)$$

En supprimant la contrainte d'intégrité du PLE , on obtient le PL dit relaxation du PLE initial

$$PL(II) \begin{cases} \min c^T x \\ s.c. \\ Ax = b \\ x \in \mathbb{R}_+^n \end{cases} \quad (1.3)$$

On résout le PL obtenu par la méthode du simplexe (voir section 1.3.1) pour aboutir une solution optimale x^* .

Si tous les x_i^* , avec $i \in \{1, 2, \dots, n\}$, sont entières, alors x^* correspond à x' la solution optimale du PLE , sinon on ajoute encore des contraintes puis on continue le processus ci-haut.

Remarque 1.3.1. *La méthode des coupes de Gomory nous permet de résoudre un PLE par usage de la méthode du simplexe, mais son principal inconvénient est que pour des problèmes raisonnables, l'algorithme peut converger parfois d'une manière trop lente vers la solution optimale.*

1.3.4 La méthode de Branch & Cut

La méthode **Branch & Cut** est une combinaison de la méthode de Branch and Bound et de la méthode de coupes de Gomory. Cette méthode améliore l'inefficacité et le manque de performance de ces deux méthodes face à certains problèmes appartenant à la classe

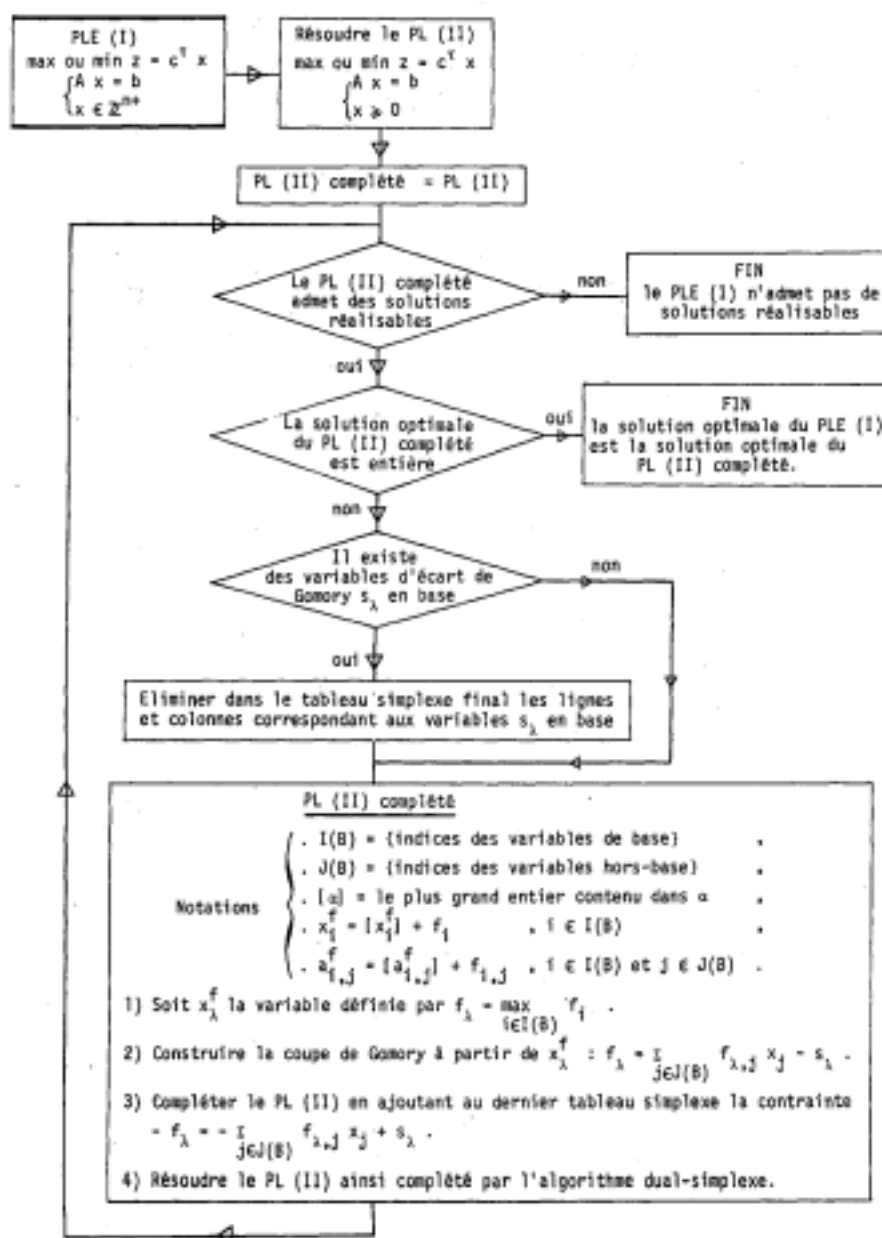


FIGURE 1.2 – Structure de la methode de Gomory, (cf support de cours : PE, p.164)

NP-difficile. Pour résoudre un *PLE*, la méthode de **Branch & Cut** commence d'abord par relaxer le problème puis appliquer les coupes planes sur la solution trouvée. Si on obtient une solution non entière, le problème sera divisé en sous-problèmes qui seront résolus de la même manière.

On se propose de résoudre le problème d'optimisation suivant par l'algorithme [1] :

$$(\min c^T x : Ax \geq b, x \in \mathbb{R}^n)$$

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

Algorithme 1 BRANCH AND CUT

Liste des problèmes = *vide*

Initialiser : le programme linéaire par le sous problème de contraintes

(A_1, b_1) avec $A_1 \in \mathbb{R}^{m_1 \times n}$ et $b_1 \in \mathbb{R}^{m_1}$ avec $m_1 \ll m$;

Étapes d'évaluation d'un sous problème :

Calculer la solution optimale \bar{x} du programme linéaire $c^T \bar{x} = \min(c^T x : A_1 x \geq b_1, x \in \mathbb{R}^n)$;

Solution courante = Appliquer la méthode des coupes planes() ;

Fin étapes d'évaluation.

Si la solution courante est réalisable **Alors**

$x^* = \bar{x}$ est la solution optimale de $\min(c^T x : Ax \geq b, x \in \mathbb{R}^n)$;

Sinon

Ajouter le problème dans Liste des sous problèmes ;

Fin du Si

Tant Que Liste des sous problèmes \neq *vide* **Faire**

Sélectionner un sous problème ;

Brancher le problème ; Appliquer les étapes d'évaluation

Fin du Tant Que

Simulation numérique de [1]

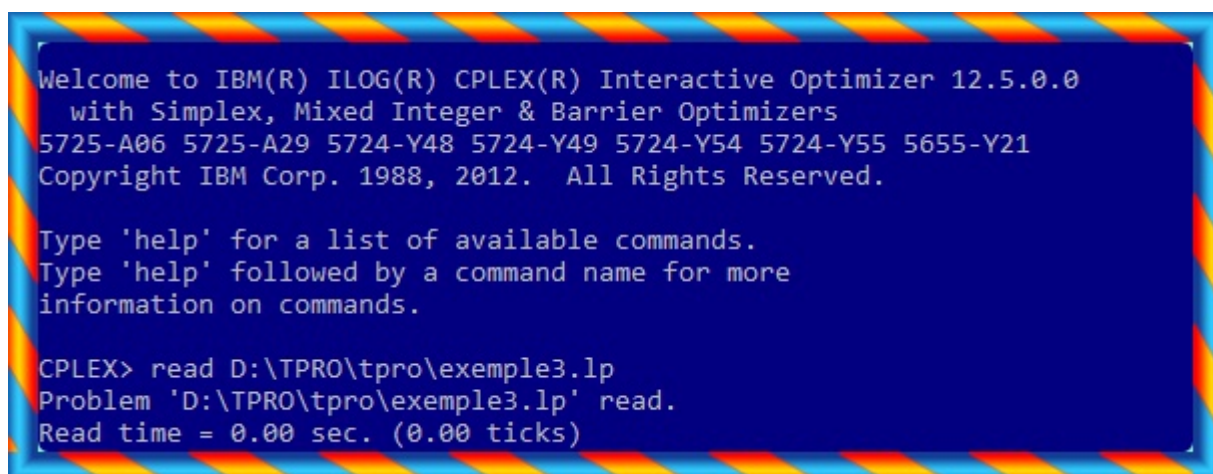
L'algorithme de **Branch and Cut** est utilisé dans base du logiciel **IBM(R) ILOG(R) CPLEX(R) optimizer version 12.5.0.0** comme on va le voir sur le travail pratique suivant :

$$(PL) \begin{cases} \max(20x_1 + 8x_2 + 6x_3 + 5x_4 + 4x_5x_6) \\ s.c. \\ 9x_1 + 8x_2 + 6x_3 + 5x_4 + 4x_5x_6 \leq 12 \\ x_i \in \{0, 1\} \text{ avec } i \in \{1, 2, 3, 4, 5, 6\} \end{cases}$$

On écrit le *PL* sous l'extension **.lp** voir (1.4). On passe à la résolution du *PL* par le Cplex puis on obtient le résultat suivant sur son terminal :

```
\Problem name: exemple3.lp

Maximize
20 x1 + 8 x2 +6 x3 + 5 x4 + 4 x5 + x6
Subject To
c1: 9x1 + 8x2 + 6x3 + 5 x4 + 4x5 + x6 <= 12
BINARY
x1
x2
x3
x4
x5
x6
End
```

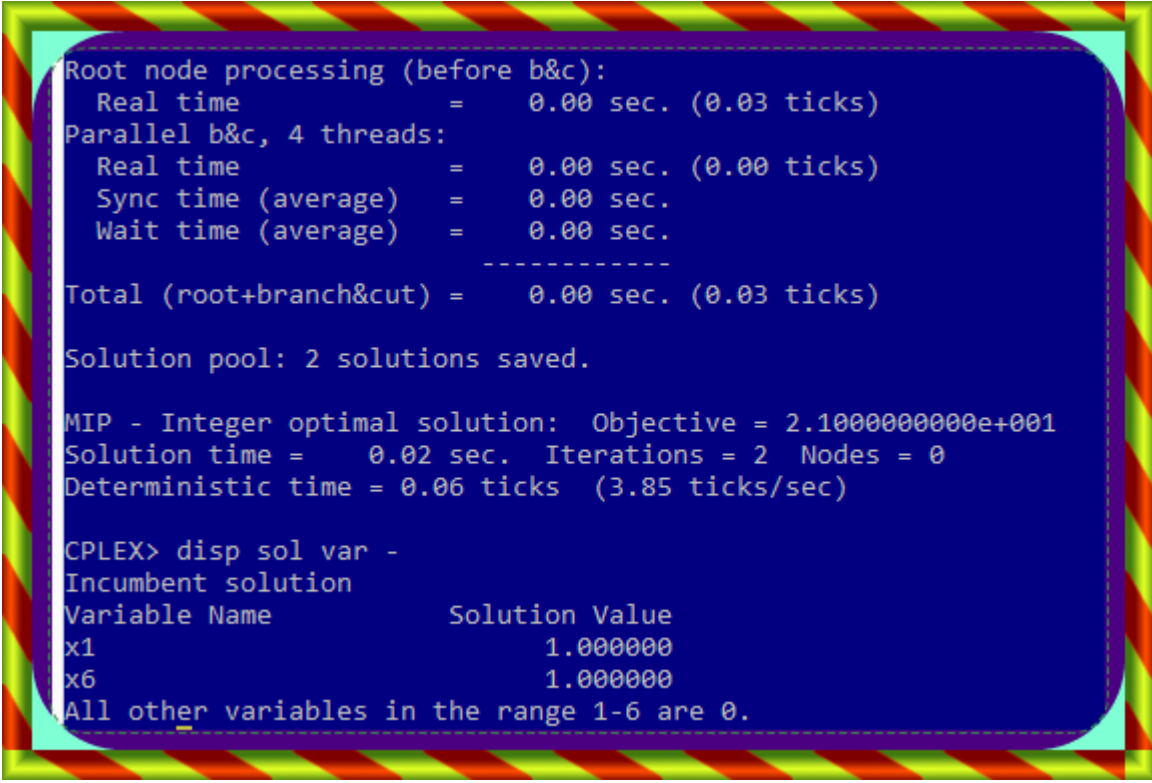
A screenshot of the CPLEX Interactive Optimizer interface. The window has a blue background with a yellow and red striped border. The text inside shows the welcome message, version information, and the command to read a problem file. The command 'CPLEX> read D:\TPRO\tpro\exemple3.lp' has been entered, and the response indicates the file was read successfully. The read time is shown as 0.00 seconds.

```
Welcome to IBM(R) ILOG(R) CPLEX(R) Interactive Optimizer 12.5.0.0
with Simplex, Mixed Integer & Barrier Optimizers
5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
Copyright IBM Corp. 1988, 2012. All Rights Reserved.

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX> read D:\TPRO\tpro\exemple3.lp
Problem 'D:\TPRO\tpro\exemple3.lp' read.
Read time = 0.00 sec. (0.00 ticks)
```

FIGURE 1.3 – Lecture du PL au format “.lp” par Cplex.



```

Root node processing (before b&c):
  Real time           =    0.00 sec. (0.03 ticks)
Parallel b&c, 4 threads:
  Real time           =    0.00 sec. (0.00 ticks)
  Sync time (average) =    0.00 sec.
  Wait time (average) =    0.00 sec.
  -----
Total (root+branch&cut) =    0.00 sec. (0.03 ticks)

Solution pool: 2 solutions saved.

MIP - Integer optimal solution: Objective = 2.1000000000e+001
Solution time =    0.02 sec. Iterations = 2  Nodes = 0
Deterministic time = 0.06 ticks (3.85 ticks/sec)

CPLEX> disp sol var -
Incumbent solution
Variable Name          Solution Value
x1                      1.000000
x6                      1.000000
All other variables in the range 1-6 are 0.

```

FIGURE 1.4 – Détails de résolution et affichage du résultat.

Remarque 1.3.2. *Il existe bien d'autres méthodes de résolution exactes des problèmes d'optimisation comme la Programmation dynamique, la méthode de résolution par Colonies etc. A ce groupe s'ajoute les méthodes heuristiques que nous allons voir en détails dans la section suivante.*

1.4 Les heuristiques

En Optimisation combinatoire, une **heuristique** est un algorithme d'approximation qui permet d'identifier en temps polynomial au moins une solution réalisable rapidement, mais obligatoirement optimale. L'usage d'une heuristique est efficace pour Calculer une solution approchée d'un problème et ainsi accélérer le processus de résolution exacte. Généralement une heuristique est conçue pour un problème particulier en s'appuyant sur sa structure propre sans offrir aucune garantie quant à la qualité de la solution Calculée. On distingue deux catégories d'heuristiques :

- ✦ Méthodes constructives qui génèrent des solutions à partir d'une solution initiale en essayant d'en ajouter petit à petit des éléments jusqu'à ce qu'une solution complète soit obtenue.
- ✦ Méthodes de fouilles locales qui démarrent avec solution initialement complète (probablement moins intéressante) et de manière répétitive essaient d'améliorer cette solution en explorant son voisinage.

1.4.1 La descente recursive (recherche locale)

La méthode de descente est l'une des heuristiques classiques les plus connues. C'est un exemple typique de recherche locale, elle progresse à travers l'ensemble des solutions X par le choix de la meilleure solution voisinage de la courante et ainsi de suite ; ce processus s'interrompt dès que le premier minimum local est atteint. Pour un problème de minimisation d'une fonction f , l'algorithme de la descente peut être décrit comme suit :

Algorithme 2 DESCENTE RECURSIVE

Solution inintial s ,

Répéter :

Si $f(s') < f(s)$ **Alors** $s := s'$,

Jusqu'à ce que $f(s') \geq f(s), \forall s' \in N(s)$.

Remarque 1.4.1. *Cette heuristique est caractérisée par sa simplicité mais présente deux inconvénients :*

- *Suivant la taille et la structure du voisinage $N(s)$ considéré, la recherche de la meilleure solution voisine qui peut être aussi difficile que le problème initial,*
- *Elle est incapable de progresser au delà du premier minimum local rencontré. Par contre les problèmes d'optimisation combinatoires comportent en générale plusieurs optima locaux pour lesquels la valeur de la fonction objectif peut être fort éloignée,*
- *La descente recursive est la méthode de recherche locale la plus élémentaire donc elle est de trajectoire.*

1.5 Conclusion

Nous avons vu que comme les méthodes exactes, les heuristiques classiques ne sont pas très satisfaisantes pour résoudre efficacement en temps polynomial les problèmes d'optimisation présentant une certaine complexité. Les solutions issues de ces méthodes ne garantissent point la qualité de la solution Calculée. Ains les chercheurs s'inspirent de la nature pour créer de nouvelles méthodes plus générales et plus efficaces comme les métaheuristiques.

LES MÉTAHEURISTIQUES

2.1 Introduction

Définitions

Classification

2.2 Méthodes à base d'une seule solution

2.2.1 Le Recuit simulé

2.2.2 La recherche Tabou

2.3 Methodes à base de populations de solutions

2.3.1 Les colonies des fourmis

2.3.2 Les algo-génétiques

2.3.3 L'Optimisation par essaim de particule

2.3.4 La recherche dispersée

2.4 Conclusion

EXEMPLES PRATIQUES D'APPLICATIONS

3.1 Introduction

3.2 Example 1

3.3 Example 2

3.4 Example 3

3.5 Conclusion

Conclusion générale

Bibliographie

- [1] Acher J. ; Gadelle J : *Programmation Linéaire*. Dunod Décision, Bordas, Paris, 1978, Troisième édition .
- [2] Cea J. : *Optimisation théorique et Algorithmes*. Dunod, Paris, 1971.
- [3] Minoux M. : *Programmation linéaire, théorie et algorithmes*. Dunod, Tomes 1 et 2
- [4] Werra D. : *Eléments de programmation linéaire avec application aux graphes*. Presse polytechniques, 1990, Première édition.
- [5] Maurras Jean F. : *Programmation linéaire, Complexité : Séparation et Optimisation*. Springer-Verlag Berlin, Heidelberg, 2002
- [6] Schrijver A. *Theory of linear and integer programming*. Wiley and Sons, 1986, (Cité page 6.)
- [7] coursC2SI.pdf
<http://www.fsr.ac.ma/cours/maths/bernoussi/Cours%20C2SI.pdf>
- [8] Revue d'Intelligence Artificielle, Vol : No.1999,
<http://www.info.univ-angers.fr/pub/hao/papers/RIA.pdf>
- [9] Thèse : conception des métaheuristiques d'optimisation,
<https://tel.archives-ouvertes.fr/tel-01143778/document>
- [10] THESE de ALAMI,
http://homepages.laas.fr/elbaz/These_LALAMI.pdf
- [11] Interstices.info : Le problème de sac à dos,
https://interstices.info/jcms/c_19213/le-probleme-du-sac-a-dos
- [12] Algorithme de cutting plane Coupe de Gomory - LITA
<http://www.lita.univ-lorraine.fr/~kratsch/teaching/ro10.ps>
- [13] Integer programming : cutting planes, (à partir de la page 301).
<http://http://web.mit.edu/15.053/www/AMP-Chapter-09.pdf>
- [14] REPERE2011,
Ressources Électroniques Pour les Etudiants, la Recherche et l'Enseignemen.
Paris, à jour le 01/06/2011, disponible sur :
<http://repere.enssib.fr>
- [15] Fouad Bekkari, [*Mémoire de Magistère*] : *Résolution des problèmes difficiles par optimisation distribuée* UNIVERSITE MOHAMED KHIDER BISKRA, Algérie, 2008.