Enseignement de l'informatique

POLYCOPIE

Méthodes d'optimisation

LEBBAH Yahia
Professeur, Université d'Oran 1

Département Informatique, Faculté des Sciences Exactes et Appliquées, Université Oran1
B.P. 1524, El-M'Naouar Oran, Algérie

Version du 29 octobre 2025

Table des matières

Chapitre 1

Introduction

Chapitre 2

Préliminaires sur l'optimisation combinatoire : problèmes faciles et problèmes difficiles[Prins, 1994]

2.1 Problèmes faciles

Exploration d'un graphe Donnée : Un graphe orienté $G = (X, U)$, deux sommets s et t de X. Question : Existe-t-il un chemin de s à t ? Algorithme en $O(M)$.

Chemin de coût minimal Données : $G = (X, U, C)$ un graphe orienté valué, et deux sommets s et t de X. Question : Trouver un chemin de coût minimal de s à t. Algorithme de Bellman en $O(NM)$. Algorithme de Dijkstra en $O(N^2)$.

Flot maximal Données : Un réseau de transport $G = (X, U, C, s, t)$. Question : Maximiser le débit du flot qui peut s'écouler dans le réseau entre s et t. Algorithme de Ford-Fulkerson en $O(NM^2)$.

Arbre recouvrant de poids minimal Données : $G = (X, E, W)$ un graphe simple valué. Question : Trouver un arbre recouvrant de poids minimal. Algorithme de Prim en $O(N^2)$. Algorithme de Kruskal en $O(M \log N)$. Si G est orienté, on pourrait calculer une arborescence recouvrante en $O(MN)$ avec l'algorithme de Edmonds.

Couplage Données : Soit $G = (X, E)$ un graphe simple. Un couplage de G (matching) est un sous-ensemble d'arêtes tel que deux quelconques d'entre elles n'aient aucun sommet commun. Question : Trouver un couplage de cardinalité maximale.

Parcours eulériens et chinois Données : Un parcours eulérien passe une fois par chaque arc ou arête. Le problème d'existence est solvable en $O(M)$. Un parcours chinois visite au moins une fois chaque arête.

Test de planarité Données : Un graphe simple $G = (X, E)$. Question : G est-il planaire, c'est-à-dire dessinable dans le plan sans croisement d'arêtes ? Un algorithme en $O(M)$ dû à Hopcroft et Tarjan.

Test de bipartisme On peut démontrer qu'un graphe est biparti ssi ne contient pas de cycle impair. Algorithme en $O(M)$.

Recherche d'une information parmi N Il s'agit de la recherche d'un élément dans un tableau de N éléments.

Tri de N **nombre** Solvable avec l'algorithme du tri par tas en $O(n \log n)$.

Programmation linéaire Il s'agit de résoudre le problème d'optimisation :

$$\begin{cases} min\ c.x \\ A.x \leq b \\ x \in I\!R^n, x \geq 0 \end{cases}$$

L'algorithme du simplexe est performant en moyenne, mais exponentiel dans le pire des cas. Karmarkar a proposé en 1984 un algorithme polynomial en $O(n^{3.5}L)$ où L est le nombre de bits pour coder A, b et c.

2.2 Problèmes difficles

Stable maximal Données : Un graphe simple $G = (X, E)$. Un sous-ensemble de sommets S est un ensemble stable s'il n'y a pas d'arête entre deux sommets quelconques de S.

Transversal minimal Données : Un graphe simple $G = (X, E)$. Un sous-ensemble de sommets T est un Transversal si toute arête de G a au moins une extrémité dans T.

Clique maximale Données : Un graphe simple $G = (X, E)$. Un sous-ensemble de sommets Q est une clique si toute paire de sommets de Q est reliée par une arête. Q engendre donc un graphe complet.

Coloration minimale Données : Un graphe simple $G = (X, E)$. G est k-colorable si on peut colorer ses sommets avec k couleurs distinctes, sans que deux sommets voisins aient la même couleur. Le plus petit k pour lequel G est k-colorable est le nombre chromatique de G.

Problèmes hamiltoniens Données : $G = (X, U, C)$ un graphe orienté valué. Le problème d'existence d'un parcours hamiltonien dans un graphe G est difficile. Le problème du voyageur de commerce consiste à trouver un circuit ou un cycle hamiltonien, de coût minimal, dans un graphe valué complet.

Problème SAT Données : une formule clausale. Question : Peut-on affecter à chaque variable propositionnelle de façon à rendre toutes les clauses vraies.

Sac à dos en variables entières Il s'agit de résoudre le problème :

$$\begin{cases} min\ c.x \\ a.x \leq b \\ x \text{ entier} \end{cases}$$

Bin packing On donne n objets de poids a_i et un nombre non limité de boites de capacité b. Le but est de répartir les objets en un nombre minimal de boites.

2.3 NP-complétude et les classes P et NP [Prins, 1994]

Certains problèmes d'optimisation combinatoire disposent d'algorithmes polynomiaux, tandis que d'autres n'en ont toujours pas. Existe-t-il réellement une classe de problèmes combinatoires pour lesquels on ne trouvera jamais d'algorithmes polynomiaux, ou est ce que les problèmes difficiles ont en fait de tels algorithmes, mais non encore découverts ? On conjecture depuis longtemps l'existence d'une classe de problèmes intrinsèquement difficiles, car plusieurs problèmes difficiles (comme le problème du voyageur de commerce) résistent depuis plus de 50 ans à l'assaut des travaux de recherche : malgré ces efforts, aucun algorithme polynomial n'a été trouvé.

La théorie de la complexité a été développée dans les années 1970 pour répondre à cette question. Le principal résultat est que tous les problèmes difficiles sont liées : la découverte d'un algorithme polynomial pour un seul problème difficile permettrait de déduire des algorithmes polynomiaux pour tous les autres.

La théorie de la complexité ne traite que des problèmes d'existence, à réponse oui/non. Ceci n'est pas gênant pour les problèmes d'optimisation. Un algorithme efficace pour le problème d'existence peut être utilisé pour résoudre efficacement son problème d'optimisation associé, par une simple dichotomie sur les valeurs de la fonction objectif.

2.4 La classe P et NP [Prins, 1994, Cori et al., 2001]

Etant donné une fonction $f : \mathbb{N} \rightarrow \mathbb{N}$, on dira qu'un problème appartient à la classe $DTIME(f)$ s'il existe une machine déterministe qui sur tout entrée de longueur n, résout le problème en $O(f(n))$ pas de calcul.

L'ensemble des problèmes d'existence qui admettent des algorithmes polynomiaux forment la classe P. $On peut poser P = \cup_{k\geq 0} DTIME(n \longmapsto n^k)$**.** Il faut pouvoir vérifier en temps polynomial une proposition de réponse Oui.

La classe NP est celles des problèmes d'existence dont une proposition de solution Oui est vérifiable polynomialement. On définit également la classe $NP = \cup_{k\geq 0} NTIME(n \longmapsto n^k)$, où N signifie non-déterministe. Le modèle de calcul non-déterministe est enrichi par une instruction de "choix" où il existe une façon immédiate pour conduire à la solution.

Les problèmes qui ne sont pas dans NP existent, mais ne présentent qu'un intérêt théorique pour la plupart. NP inclut P.

Pour un problème sans algorithme efficace, il faut procéder comme suit pour prouver l'appartenance à NP :

1. proposer un codage de la solution (appelé certificat) ;
2. proposer un algorithme qui va vérifier la solution au vu des données du certificat ;
3. montrer que cet algorithme a une complexité polynomiale.

Considérons le problème suivant : **étant donné un ensemble S de n nombres entiers et un entier b, existe-t-il un sous-ensemble T de S dont la somme des éléments est égale à b ?** On ne connait pas d'algorithme polynomial pour résoudre ce problème. Il n'empêche qu'il est dans NP, car **vérifier qu'une somme d'un ensemble d'entiers T, sous-ensemble d'un ensemble S de cardinalité n, est égale à b, est en $O(n^2)$**. Dans cette vérification, il faut s'assurer que tous les éléments de T sont dans S, qui nécessitera au plus n^2 tests.

2.5 Les problèmes NP-complets [Prins, 1994, Cori et al., 200

Il s'agit des problèmes les plus difficiles de NP, le "noyau dur". La notion de problème NP-complet est basée sur celle de transformation polynomiale d'un problème. Un problème d'existence P_1 se transforme polynomialement en un autre P_2 s'il existe un algorithme polynomial A transformant toute donnée pour P_1 en une pour P_2, en conservant la réponse Oui et Non. Par exemple, un stable d'un graphe simple G est une clique dans le graphe complémentaire G_c.

Un problème NP-complet est un problème de NP en lequel se transforme polynomialement tout autre problème de NP. La classe des problèmes NP-complets est notée NPC. On rencontre dans la littérature le terme NP-difficile pour les problèmes d'optimisation : un problème d'optimisation combinatoire est NP-difficile si le problème d'existence associé est NP-complet.

La conséquence pratique forte de la classe NPC : si on trouverait un algorithme polynomial A pour un seul problème NP-complet X, on pourrait en déduire un autre pour tout autre problème difficile Y de P. Il suffit de transformer polynomialement les données de Y en données pour X, puis exécuter l'algorithme pour X.

Une question immédiate est de savoir si de tels problèmes existent réellement dans NP. Le logicien Cook (et Levin) a montré en 1970 que **le problème SAT est NP-complet**. Depuis cette date, les travaux de recherche ont montré que la plupart des problèmes d'existence associés aux problèmes difficiles sans algorithmes polynomiaux connus sont NP-complets.

Problème SAT

Données Soit un ensemble de de variables $\{x_1, ..., x_n\}$. Soit une formule logique F sous forme normale conjonctive $F = C_1 \wedge C_2 \wedge ... \wedge C_l$, avec chaque clause $C_i = (y_{i,1} \vee y_{i,2} \vee ... \vee y_{i,k_i})$, où chaque $y_{i,j}$ est un littéral, c'est-à-dire $y_{i,j} = x_i$ ou $y_{i,j} = \neg x_i$.

Résultat "oui" ssi F est satisfaisable, qu'il existe une affectation de valeurs de vérités aux variables qui rende F vraie.

Proposition 1. *Le problème SAT est NP-complet.*

Preuve : Premier problème démontré NP-complet. Preuve réalisée par Cook et Levin [Cook, 1971, Levin, 1973]. □

Chapitre 3

Optimisation linéaire et résolution exacte par séparation/évaluation

Les sections 3.1, 3.3, 3.2 sont extraits intégralement du document [Bastin, 2010] téléchargeable ici : https://www.iro.umontreal.ca/~bastin/Cours/IFT1575/IFT1575.pdf

3.1 Présentation générale [Bastin, 2010]

Certaines quantités ne peuvent s'écrire sous forme de nombres réels, issus d'un domaine continu. Au contraire, certaines décisions sont par nature discrètes, et doivent se représenter à l'aide de nombres entiers. Considérons par exemple une **entreprise de transport**, qui décide de renouvellement sa flotte de camions. Le **nombre de camions à acheter est un nombre naturel**.

La présence de telles variables entières modifie profondément la nature des programmes sous-jacents. Lorsqu'un problème est linéaire avec des variables entières, nous parlerons de *programmation mixte entière*. Si toutes les variables sont entières, nous utiliserons la terminologie de de *programmation (pure) en nombres entiers*. Si les variables entières sont à valeurs 0 ou 1 (binaires), nous parlerons de **programmation 0–1 (binaire)**.

Nous pourrions bien entendu considérer le cas **non-linéaire**, mais les complications sont telles qu'à ce jour, **aucune méthode pleinement satisfaisante n'existe**, même si d'intenses recherches sont actuellement conduites à ce niveau, notamment au sein de l'entreprise IBM.

Exemple 1 (Problème du sac à dos)**.** *Considérons un transporteur muni d'un sac (unique) pour transporter son butin. Son problème consiste à* **maximiser la valeur totale des objets** *qu'il emporte, sans toutefois dépasser une limite de* **poids** b *correspondant à ses capacités physiques. Supposons qu'il y a* n **type d'objets** *que le transporteurpourrait emporter, et que ceux-ci sont en nombre tel que quelle que soit la nature de l'objet considéré, la seule limite au nombre d'unités que le transporteur peut prendre est que le* **poids total reste inférieur à** b*. Si l'on associe* **à l'objet** j **une valeur** c_j **et un poids** w_j*, la* **combinaison optimale** *d'objets à emporter*

sera obtenue en résolvant le programme mathématique

$$\begin{aligned} \max_x \ & \sum_{j=1}^{n} c_j x_j \\ s.c. \ & \sum_{j=1}^{n} w_j x_j \leq b \\ & x_j \in \mathcal{N}, \ j = 1, \ldots, n. \end{aligned}$$

Ici, $x_j \in \mathcal{N}$ **signifie que** x_j **est un naturel**, *autrement dit un entier non négatif.* **Intuitivement, nous pourrions penser que la solution consiste à choisir en premier lieu les objets dont le rapport qualité-poids est le plus avantageux**, *quitte à* **tronquer** *pour obtenir une solution entière (nous ne pouvons pas diviser un objet). Cette solution peut malheureusement se révéler* **sous-optimale, voire mauvaise**, *comme on le constate sur l'exemple suivant.*

$$\begin{aligned} \max_x \ & 2x_1 + 3x_2 + 7x_3 \\ s.c. \ & 3x_1 + 4x_2 + 8x_3 \leq 14, \\ & x_1, \ x_2, \ x_3 \in \mathcal{N}. \end{aligned}$$

Si nous oublions la contrainte d'intégralité, la solution, de valeur 12.25, *est* $x = (0, 0, 14/8)$. *En* **tronquant**, *on obtient la solution entière* $x = (0, 0, 1)$ **de valeur 7**. *Il est facile de trouver de meilleures solutions, par exemple* $x = (1, 0, 1)$.

La solution optimale du problème du transporteur peut s'obtenir en **énumérant toutes les solutions admissibles** *et en conservant la meilleure (voir Table 3.1, , où les solutions inefficaces laissant la possibilité d'ajouter un objet n'ont pas été considérées).*

x_1	x_2	x_3	objectif
0	1	1	10
2	0	1	11
0	3	0	9
2	2	0	10
3	1	0	9
4	0	0	8

TABLE 3.1 – Problème du sac à dos : énumération des solutions

La solution optimale entière, $x = (2, 0, 1)$, **diffère** *passablement de la solution optimale linéaire. Cependant, il est clair que cette technique d'énumération* **ne peut s'appliquer à des problèmes de grande taille**.

Exemple 2. *Une entreprise doit* **choisir de nouveaux emplacements pour construire des usines et des entrepôts**. *Elle a le choix entre deux emplacements : Oran (LA) et Constantine (SF). Nous ne pouvons construire un entrepôt que dans une* **ville où nous disposons d'une usine**, *et nous ne pouvons* **pas**

	Valeur estimée (millions $)	Coût de construction (millions $)
Usine à LA	9	6
Usine à SF	5	3
Entrepôt à LA	6	5
Entrepôt à SF	4	2
Limite maximum	-	10

TABLE 3.2 – Données du problème

construire plus d'un entrepôt. *Nous associons à chaque construction (d'une usine ou d'un entrepôt dans chacun des lieux envisagés)* **sa valeur estimée et son coût** *de construction. L'objectif est de* **maximiser la valeur totale estimée**, *en ne dépassant pas une limite maximum sur les coûts. Les données du problème sont résumées dans la Table 3.2. Les* **variables** *sont*

$$x_j = \begin{cases} 1 & \text{si la décision } j \text{ est approuvée ;} \\ 0 & \text{si la décision } j \text{ n'est pas approuvée.} \end{cases}$$

L'objectif est de **maximiser la valeur estimée totale** *:*

$$\max z = 9x_1 + 5x_2 + 6x_3 + 4x_4.$$

Les contraintes fonctionnelles sont

1. *la* **limite maximum sur les coûts** *de construction :*

$$6x_1 + 3x_2 + 5x_3 + 2x_4 \leq 10;$$

2. *nous ne pouvons construire* **plus d'un entrepôt** *:*

$$x_3 + x_4 \leq 1;$$

3. *l'entrepôt ne sera* **à LA que si l'usine est à LA** *:*

$$x_3 \leq x_1;$$

4. *l'entrepôt* **ne sera à SF que si l'usine est à SF** *:*

$$x_4 \leq x_2 :$$

5. *contraintes 0–1 (intégralité) :*

$$x_j \in \{0, 1\}, \ j = 1, 2, 3, 4;$$

ou encore

$$0 \leq x_j \leq 1 \text{ et } x_j \text{ entier}, \ j = 1, 2, 3, 4.$$

Par conséquent, nous avons le **programme mathématique**

$$\begin{aligned} max\ z &= 9x_1 + 5x_2 + 6x_3 + 4x_4 \\ s.c.\ & 6x_1 + 3x_2 + 5x_3 + 2x_4 \leq 10; \\ & x_3 + x_4 \leq 1; \\ & -x_1 + x_3 \leq 0; \\ & -x_2 + x_4 \leq 0; \\ & x_1, x_2, x_3, x_4 \leq 1; \\ & x_1, x_2, x_3, x_4 \geq 0; \\ & x_1, x_2, x_3, x_4 \ entiers. \end{aligned}$$

Le modèle illustre deux cas classiques d'illustrations de **variables binaires** *:*

- *alternatives* **mutuellement exclusives** *: nous ne pouvons construire plus d'un entrepôt, i.e.*

$$x_3 + x_4 \leq 1;$$

- **décisions contingentes** *: nous ne pouvons construire un entrepôt que là où nous avons construit une usine :*

$$x_3 \leq x_1; x_4 \leq x_2 :$$

Exemple 3 (Localisation)**.** *Une entreprise envisage plusieurs sites de construction pour des usines qui serviront à approvisionner ses clients. A chaque site potentiel i correspond* **un coût de construction** a_i*, une* **capacité de production** u_i*, un* **coût de production unitaire** b_i *et des* **coûts de transport** c_{ij} *des usines vers les clients. Soit* y_i **une variable binaire** *prenant la valeur 1 si un entrepôt est construit sur le site i,* d_j **la demande de l'usine** j *et* x_{ij} **la quantité produite à l'usine** i *et destinée au marché j (flot de i à j). Un plan de construction optimal est obtenu en résolvant le programme*

$$\begin{aligned} \min_x\ & \sum_i a_i y_i + \sum_i b_i \sum_j x_{ij} + \sum_i \sum_j c_{ij} x_{ij} \\ s.c.\ & \sum_i x_{ij} = d_j, \\ & \sum_j x_{ij} \leq u_i y_i, \\ & x_{ij} \geq 0, \\ & y_i \in \{0, 1\}. \end{aligned}$$

Cette formulation contient deux éléments intéressants : un **coût fixe** *(construction) modélisé par une variable binaire y_i ainsi qu'une* **contrainte logique** *forçant les flots provenant d'un site à être nuls si aucune usine n'est construite en ce site. Notons aussi que certaines variables sont entières alors que* **d'autres (flots) sont réelles**.

Exemple 4 (**Contraintes logiques**)**. Des variables binaires peuvent servir à représenter des contraintes logiques**. *En voici quelques exemples, où* p_i **représente une proposition logique** *et x_i la variable logique (binaire)* **correspondante**.

contrainte logique	**forme algébrique**
$p_1 \oplus p_2 = vrai$	$x1 + x2 = 1$
$p_1 \vee p_2 \vee \cdots \vee p_n = vrai$	$x_1 + x_2 + \ldots + x_n \geq 1$
$p_1 \wedge p_2 \wedge \ldots \wedge p_n = vrai$	$x_1 + x_2 + \ldots + x_n \geq n$ $(ou = n)$
$p_1 \Rightarrow p_2$	$x_2 \geq x_1$
$p_1 \Leftrightarrow p_2$	$x_2 = x_1$

Exemple 5 (**Fonctions linéaires par morceaux**). *Considérons une fonction objectif à maximiser, pour laquelle dans chaque* **intervalle** $[a_{i-1}, a_i]$ **la fonction est linéaire**, *ce que nous pouvons exprimer par :*

$$\begin{aligned} &x = \lambda_{i-1} a_{i-1} + \lambda_i a_i \\ &\lambda_{i-1} + \lambda_i = 1, \\ &\lambda_{i-1},\ \lambda_i \geq 0, \\ &f(x) = \lambda_{i-1} f(a_{i-1}) + \lambda_i f(a_i). \end{aligned}$$

Car, il est bien établi que les points d'un segment d'extrémités A et B :

$$[A, B] = \{(1-t)A + tB | t \in [0, 1]\}.$$

Nous pouvons généraliser cette formule sur tout l'intervalle de définition de la fonction f **en contraignant les variables λ_i à ne prendre que deux valeurs non nulles, et ce pour deux indices consécutifs**. *Ceci se fait en introduisant des variables binaires y_i associées aux* **intervalles de linéarité** $[a_{i-1}, a_i]$ *:*

$$\begin{aligned} x &= \sum_{i=0}^{n} \lambda_i a_i, \\ f(x) &= \sum_{i=0}^{n} \lambda_i f(a_i), \\ \sum_{i=0}^{n} \lambda_i &= 1, \\ \lambda_i &\geq 0,\ i = 0, \ldots, n \\ \lambda_0 &\leq y_1, \\ \lambda_1 &\leq y1 + y2, \\ &\vdots \quad \vdots \quad \vdots \\ \lambda_{n-1} &\leq y_{n-1} + y_n, \\ \lambda_n &\leq y_n, \\ \sum_{i=1}^{n} y_i &= 1 (\textit{un seul intervalle “actif”}) \\ y_i &\in 0, 1,\ i = 1, \ldots, n. \end{aligned}$$

3.2 Contraintes mutuellement exclusives [Bastin, 2010]

3.2.1 Deux contraintes

Prenons l'exemple de deux contraintes. **L'une ou l'autre des deux contraintes doit être satisfaite, mais pas les deux simultanément.** Par exemple,

— soit $3x_1 + 2x_2 \leq 18$;
— soit $x_1 + 4x_2 \leq 16$.

Soit M **un très grand nombre** ; le système précédent **est équivalent à**

— soit

$$\begin{aligned} 3x_1 + 2x_2 &\leq 18, \\ x_1 + 4x_2 &\leq 16 + M; \end{aligned}$$

— soit

$$\begin{aligned} 3x_1 + 2x_2 &\leq 18 + M, \\ x_1 + 4x_2 &\leq 16. \end{aligned}$$

En introduisant une variable binaire y, nous obtenons le **système équivalent**

$$\begin{aligned} 3x_1 + 2x_2 &\leq 18 + M(1 - y), \\ x_1 + 4x_2 &\leq 16 + My. \end{aligned}$$

La signification de cette variable est

— $y = 1$, si la première contrainte est satisfaite ;
— $y = 0$, si la seconde contrainte est satisfaite.

Nous avons de la sorte construit deux alternatives mutuellement exclusives.

Nous aurions pu aussi introduire deux variables binaires :

— $y_1 = 1$, si la première contrainte est satisfaite ;
— $y_2 = 1$, si la seconde contrainte est satisfaite.

Nous devons avoir

$$y_1 + y_2 = 1.$$

Afin de se ramener au modèle précédent, il suffit de poser

$$\begin{aligned} y_1 &= y; \\ y_2 &= 1 - y. \end{aligned}$$

Il s'agit d'un cas particulier de la situation suivante : K parmi N contraintes doivent être satisfaites. Dans ce cas plus général, nous introduisons N variables binaires.

3.2.2 K contraintes parmi N

Soit les N **contraintes**

$$f_j(x_1, x_2, \ldots, x_n) \leq d_j, \ j = 1, 2, \ldots, N.$$

Nous introduisons N variables binaires, avec $y_j = 1$ **si la** j^e **contrainte est satisfaite** :

$$f_j(x_1, x_2, \ldots, x_n) \leq d_j + M(1 - y_j),\ j = 1, 2, \ldots, N.$$

Il reste à spécifier que seulement K **de ces contraintes peuvent êtres satisfaites** :

$$\sum_{j=1}^{N} y_j = K.$$

3.2.3 Fonction ayant N valeurs possibles

Soit la **contrainte**

$$f(x_1, x_2, \ldots, x_n) = d_1,\ \text{ou } d_2 \text{ ou } \ldots \text{ ou } d_N.$$

Nous introduisons N **variables binaires, avec** $y_j = 1$ **si la fonction vaut** d_j**.** La contrainte s'écrit alors

$$f(x_1, x_2, \ldots, x_n) = \sum_{j=1}^{N} d_j y_j,$$

avec

$$\sum_{j=1}^{N} y_j = 1.$$

Exemple 6 (Wyndor Glass)**.** *Supposons que le temps de production maximum à l'usine 3 n'est pas toujours 18h, mais pourrait également être 6h ou 12h. Cette contrainte s'écrit alors*

$$3x_1 + 2x_2 = 6 \text{ ou } 12 \text{ ou } 18.$$

Nous introduisons alors trois variables binaires

$$3x_1 + 2x_2 = 6y_1 + 12y_2 + 18y_3,$$
$$y_1 + y_2 + y_3 = 1.$$

3.2.4 Objectif avec coûts fixes

Supposons que le coût associé à un produit j est composé de deux parties :

1. un coût fixe initial k_j encouru dès qu'une unité de j est produite ;
2. un coût c_j proportionnel au nombre d'unités de j produites.

Le coût total associé à la production de x_j **unités est**

$$f(x_j) = \begin{cases} k_j + c_j x_j & \text{si } x_j > 0, \\ 0 & \text{si } x_j = 0. \end{cases}$$

Supposons de plus que l'objectif consiste à minimiser la **somme de** n **fonctions avec coûts fixes**

$$\min z = \sum_{j=1}^{n} f_j(x_j).$$

Nous introduisons alors n variables binaires :

$$y_j = \begin{cases} 1 \text{ si } x_j > 0, \\ 0 \text{ si } x_j = 0. \end{cases}$$

L'objectif s'écrit alors

$$\min z = \sum_{j=1}^{n} c_j x_j + k_j y_j.$$

Les valeurs de x_j et de y_j dépendent l'une de l'autre : il s'agit d'un exemple de *décisions contingentes*. Nous devons en particulier avoir une contrainte qui précise que $x_j = 0$ **si** $y_j = 0$. Toutefois, les deux variables ne sont plus binaires, vu que x_j peut être quelconque. Soit M_j **une borne supérieure sur la valeur de** x_j. Nous pouvons écrire la **relation entre les deux variables** de cette manière :

$$x_j \leq M_j y_j.$$

Ainsi,

- si $y_j = 0$, alors $x_j = 0$;
- si $y_j = 1$, alors $x_j \leq M_j$;
- si $x_j > 0$, alors $y_j = 1$;
- si $x_j = 0$, alors toute soluton optimale satisfait $y_j = 0$ lorsque $k_j > 0$ (si $k_j = 0$, la variable y_j est inutile).

Nous obtenons par conséquent le programme

$$\begin{aligned} \min\ z &= \sum_{j=1}^{n} c_j x_j + k_j y_j \\ \text{s.c. } x_j &\leq M_j y_j, \\ y_j &\in \{0,1\},\ j = 1, 2, \ldots, n. \end{aligned}$$

3.2.5 Variables entières en variables 0–1

Soit x une **variable entière générale bornée** :

$$0 \leq x \leq u,$$

et soit N l'entier **tel que** $2^N \leq u \leq 2^{N+1}$. La **représentation binaire de** x est

$$x = \sum_{j=0}^{N} 2^j y_j.$$

L'intérêt de cette transformation est que les méthodes de programmation 0–1 sont souvent plus efficaces que les méthodes de programmation en nombres entiers. Elle engendre néanmoins une **multiplication du nombre de variables**.

Exemple 7. *Nous considérons* **trois types de produits, et deux usines** *; nous exprimons le* **profit par unité de produit** *en milliers de dollars. Nous connaissons les* **ventes potentielles par produit** *(unités/semaine), et la* **capacité de**

	Produit 1 temps de production (h/unité)	Produit 2 temps de production (h/unité)	Produit 3 temps de production (h/unité)	Capacité de production production (h/semaine)
Usine 1	3	4	2	30
Usine 2	4	6	2	40
Profit/unité (1000$)	5	7	3	–
Ventes potentielles (par semaine)	7	5	9	–

TABLE 3.3 – Exemple de production avec variables entières.

production par usine *(h/semaine). Nous avons toutefois comme contrainte que* **pas plus de deux produits ne peuvent être fabriqués**, *et* **une seule des deux usines doit être exploitée**. *Les données du problème sont résumées dans la Table 3.3. Les variables sont* x_j, **le nombre d'unités fabriquées du produit** j. *Pour représenter la* **contrainte "pas plus de deux produits"**, *nous devons introduire des variables 0–1 :*

$$y_j = \begin{cases} 1 & si\ x_j > 0; \\ 0 & si\ x_j = 0. \end{cases}$$

Afin de représenter la contrainte **"une seule des deux usines"**, *nous devons ajouter une variables 0–1 :*

$$y_4 = \begin{cases} 1 & si\ l'usine\ 1\ est\ choisie; \\ 0 & si\ sinon. \end{cases}$$

*L'***objectif** *est*

$$\max z = 5x_1 + 7x_2 + 3x_3.$$

Les **ventes potentielles** *sont*

$$x_1 \leq 7,\ x_2 \leq 5,\ x_3 \leq 9.$$

L'exigence **interdisant d'avoir plus de deux produits** *se traduit mathématiquement par*

$$y_1 + y_2 + y_3 \leq 2.$$

La **relation entre les variables continues et les variables 0–1** *s'exprime par*

$$x_1 \leq 7y_1,\ x_2 \leq 5y_2,\ x_3 \leq 9y_3.$$

La contrainte portant sur l'utilisation d'une seule usine est

- *soit* $3x_1 + 4x_2 + 2x_3 \leq 30$;
- *soit* $4x_1 + 6x_2 + 2x_3 \leq 40$.

En utilisant la variable 0–1 (et M très grand), elle se traduit par

$$3x_1 + 4x_2 + 2x_3 \leq 30 + M(1 - y_4),$$
$$4x_1 + 6x_2 + 2x_3 \leq 40 + My_4.$$

En résumé, nous avons le modèle

$$\begin{aligned} \max z &= 5x_1 + 7x_2 + 3x_3 \\ s.c.\ & x_1 \leq 7y_1,\ x_2 \leq 5y_2,\ x_3 \leq 9y_3 \\ & y_1 + y_2 + y_3 \leq 2, \\ & 3x_1 + 4x_2 + 2x_3 \leq 30 + M(1 - y_4), \\ & 4x_1 + 6x_2 + 2x_3 \leq 40 + My_4, \\ & x_1, x_2, x_3 \geq 0, \\ & y_j \in \{0, 1\}, j = 1, 2, 3, 4. \end{aligned}$$

Exemple 8. **Nous considérons à nouveau trois types de produits, pour lesquels nous pouvons placer cinq annonces publicitaires, avec un maximum de trois annonces par produit.** *L'estimation des revenus publicitaires est donnée dans la Table 3.4, où les profits sont exprimés en millions de dollars. Les*

Nombre d'annonces	Produit 1	Produit 2	Produit 3
0	0	0	0
1	1	0	-1
2	3	2	2
3	3	3	4

TABLE 3.4 – Revenus publicitaires.

variables du problèmes sont le nombre d'annonces pour le produit i, dénoté par x_i, mais l'hypothèse de proportionnalité est alors violée : nous ne pouvons représenter l'objectif sous forme linéaire uniquement avec ces variables.

Prenons tout d'abord comme variables

$$y_{ij} = \begin{cases} 1 & si\ x_i = j; \\ 0 & sinon. \end{cases}$$

L'objectif est

$$\max z = y_{11} + 3y_{12} + 3y_{13} + 2y_{22} + 3y_{23} - y_{31} + 2y_{32} + 4y_{33}.$$

Nous utiliserons les **5 annonces disponibles** *:*

$$\sum_{i=1}^{3} \sum_{j=1}^{3} jy_{ij} = 5.$$

Enfin, on a droit à **un seul type d'annonce par produit, la définition des variables 0–1** *donne*

$$\sum_{j=1}^{3} y_{ij} \leq 1, \ i = 1, 2, 3.$$

Soit une autre modélisation *en prenant comme variables*

$$y_{ij} = \begin{cases} 1 & si\ x_i \geq j; \\ 0 & sinon. \end{cases}$$

Autrement dit, nous avons remplacé l'égalité dans la première condition par une inégalité. Cette définition implique

$$\begin{aligned} x_i = 0 &\Rightarrow y_{i1} = 0, y_{i2} = 0, y_{i3} = 0; \\ x_i = 1 &\Rightarrow y_{i1} = 1, y_{i2} = 0, y_{i3} = 0; \\ x_i = 2 &\Rightarrow y_{i1} = 1, y_{i2} = 1, y_{i3} = 0; \\ x_i = 3 &\Rightarrow y_{i1} = 1, y_{i2} = 1, y_{i3} = 1. \end{aligned}$$

Ce qui peut encore s'énoncer somme

$$y_{i(j+1)} \leq y_{ij}, \ i = 1, 2, 3, \ j = 1, 2.$$

Supposons que $x_1 = 3$ (il y a trois annonces pour le produit 1). *Le profit associé à cette valeur doit être 3. Mais* $x_1 = 3$ **veut aussi dire que chaque variable binaire associée au produit 1 vaut 1** *; comment dès lors comptabiliser correctement la contribution de ces trois variables au profit ?* **La solution consiste à prendre comme profit associé à la variable** y_{ij} **la différence** $c_{ij+1} - c_{ij}$, *où* c_{ij} *est le revenu net si nous plaçons* j *annonce pour le produit* i. *Dans notre exemple, le profit associé à*

- y_{11} *est 1-0 = 1 ;*
- y_{12} *est 3-1 = 2 ;*
- y_{13} *est 3-3 = 0 ;*

Nous obtenons ainsi le programme mathématique suivant :

$$\begin{aligned} \max\ & z = y_{11} + 2y_{12} + 2y_{22} + y_{23} - y_{31} + 3y_{32} + 2y_{33} \\ s.c.\ & y_{i(j+1)} \leq y_{ij}, i = 1, 2, 3, \ j = 1, 2, \\ & \sum_{i=1}^{3} \sum_{j=1}^{3} y_{ij} = 5, \\ & y_{ij} \in \{0, 1\}, \ i = 1, 2, 3, \ j = 1, 2. \end{aligned}$$

3.2.6 Problème de recouvrement

Exemple 9 (**Affectation des équipages**). *Un problème important des compagnies aériennes consiste à* **constituer de façon efficace des équipages pour ses vols**. *Pour un équipage donné, une* **rotation consiste en une succession de services de vol débutant et se terminant en une même ville**. *Comme il y a un* **coût**

associé à chaque séquence de vols, *la compagnie cherche à* **minimiser les coûts d'affectation des équipages aux séquences** *tout en assurant le service sur chacun des vols.*

Considérons par exemple un problème avec 11 vols et 12 séquences de vols, dont les données sont décrites dans la Table 3.5. Les variables sont

Vol | Séquence	1	2	3	4	5	6	7	8	9	10	11	12
1	1			1			1			1		
2		1			1			1			1	
3			1			1			1			1
4				1			1		1	1		1
5	1					1				1	1	
6				1	1				1			
7							1	1		1	1	1
8		1		1	1				1			
9					1			1			1	
10			1				1	1				1
11						1			1	1	1	1
Coût	2	3	4	6	7	5	7	8	9	9	8	9

TABLE 3.5 – Affectation d'équipages.

$$x_j = \begin{cases} 1 & \textit{si la séquence de vols } j \textit{ est affectée;} \\ 0 & \textit{sinon.} \end{cases}$$

L'objectif est

$$\min z = 2x_1 + 3x_2 + 4x_3 + 6x_4 + 7x_5 + 5x_6 + 7x_7 + 8x_8 + 9x_9 + 9x_{10} + 8x_{11} + 9x_{12}.$$

Nous devons affecter **trois équipages**

$$\sum_{j=1}^{12} x_j = 3.$$

Le service doit être **assuré sur chacun des vols** *:*

$$\begin{aligned} x_1 + x_4 + x_7 + x_{10} &\geq 1 \\ x_2 + x_5 + x_8 + x_{11} &\geq 1 \\ x_3 + x_6 + x_9 + x_{12} &\geq 1 \\ x_4 + x_7 + x_9 + x_{10} + x_{12} &\geq 1 \\ &\dots \end{aligned}$$

Généralement, un **problème de recouvrement d'ensemble** met en oeuvre
— I : un ensemble d'objets (les vols dans l'exemple précédent) ;
— $\mathcal{J}$: une collection de sous-ensembles de I (e.g. les séquences de vols) ;

— J_i, $i \in I$: les sous-ensembles dans $\mathcal{J}$ qui contiennent i.

Nous avons les variables binaires x_j, **prenant la valeur 1 si le sous-ensemble j est choisi, et 0 sinon**. En considérant un objectif linéaire, avec c_j le coût associé au sous-ensemble j. Nous obtenons le programme

$$\begin{aligned} \min_x & \sum_{j \in \mathcal{J}} c_j x_j \\ \text{s.c.} & \sum_{j \in J_i} x_j \geq 1, \ i \in I; \\ & x_j \in \{0,1\}, \ j \in \mathcal{J}. \end{aligned}$$

3.3 Stratégies de résolutions[Bastin, 2010]

3.3.1 Relaxation linéaire

Il est tentant "d'oublier" les contraintes d'intégralité, et de résoudre le problème en nombres continus ainsi produit. Nous parlerons alors de *relaxation*. Ainsi, nous pourrons construire la relaxtion en programme linéraire d'un programme mixte entier. **Une fois le programme relâché résolu, nous pourrions arrondir aux valeurs entières les plus proches.** Dans certains cas, cela peut fonctionner, mais l'exemple du sac à dos montre que **ce n'est pas toujours vérifié**. Cette méthode par arrondissement est même **parfois désastreuse**.

Exemple 10. *Considérons le programme*

$$\begin{aligned} \max \ & z = x_2 \\ s.c. \ & -x_1 + x_2 \leq \frac{1}{2}, \\ & x_1 + x_2 \leq \frac{7}{2}, \\ & x_1, x_2 \geq 0 \ \textit{et entiers}. \end{aligned}$$

La relaxation en programme linéaire donne

$$\begin{aligned} \max \ & z = x_2 \\ s.c. \ & -x_1 + x_2 \leq \frac{1}{2}, \\ & x_1 + x_2 \leq \frac{7}{2}, \\ & x_1, x_2 \geq 0. \end{aligned}$$

Ce nouveau programme a pour **solution** $\left(\frac{3}{2}, 2\right)$. *Que nous arrondissions cette solution à* $(1,2)$ *ou* $(2,2)$, **nous n'obtenons pas de solution réalisable**, *comme illustré sur la Figure 3.1*

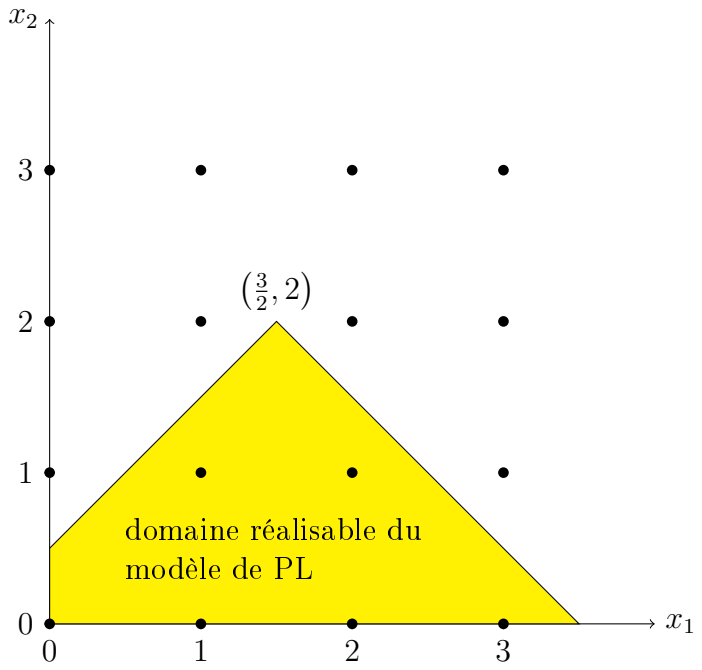


FIGURE 3.1 – Exemple de relaxation linéaire : problème d'admissibilité des solutions

Exemple 11. *Considérons le programme*

$$\begin{aligned} \max\ & z = x_1 + 5x_2 \\ s.c.\ & x_1 + 10x_2 \leq 20, \\ & x_2 \leq 2, \\ & x_1, x_2 \geq 0 \text{ et entiers.} \end{aligned}$$

La version relâchée de programme est

$$\begin{aligned} \max\ & z = x_1 + 5x_2 \\ s.c.\ & x_1 + 10x_2 \leq 20, \\ & x_2 \leq 2, \\ & x_1, x_2 \geq 0, \end{aligned}$$

qui a pour solution optimale $(2, 1.8)$. **En arrondissant à** $(2, 1)$ **afin de garantir l'admissibilité, nous obtenir la valeur 7 pour la fonction objectif, loin de la valeur optimale du programme mixte entier, avec pour valeur optimale 11**, *en* $(0, 2)$ *(voir Figure 3.2).*

La solution de la relaxation linéaire ne peut donc être exploité directement pour obtenir la solution exacte du problème en nombres entiers.

Cependant, la relaxation linéaire a les propriétés suivantes :

Borne supérieure La valeur de la solution optimale de la relaxation est une borne supérieure sur la valeur de la solution optimale du problème de maximisation en nombres entiers. C'est une borne inférieur sur un problème de minimisation.

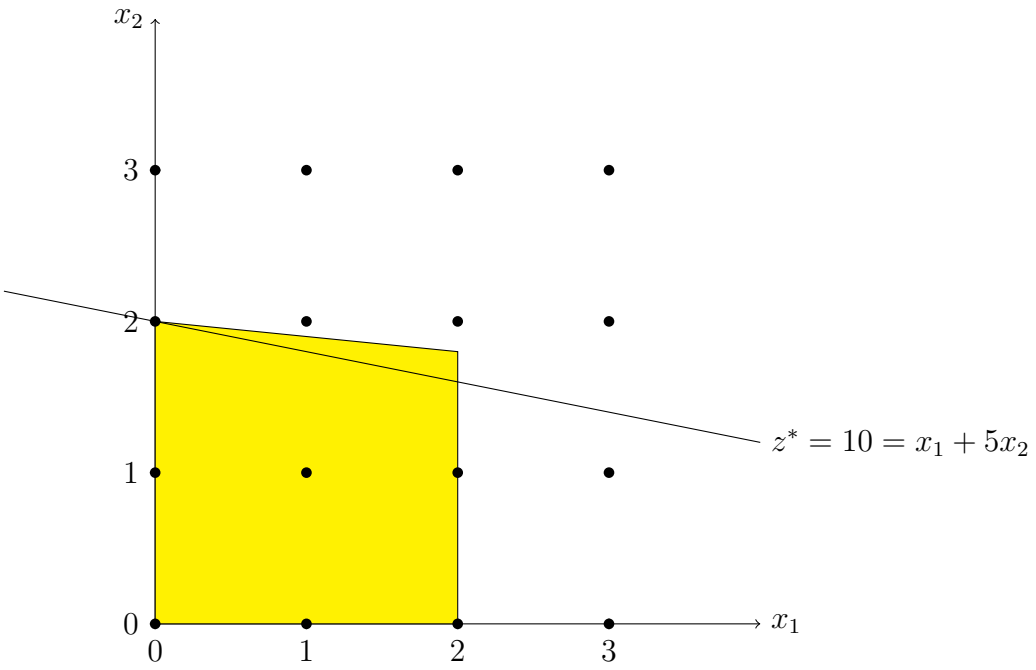


FIGURE 3.2 – Exemple de relaxation linéaire : solution médiocre

Solution entière Si la solution optimale de la relaxation est entière (donc admissible pour le problème en nombres entiers), elle est également la solution optimale du problème en nombres entiers.

Borne inférieure La valeur d'une solution admissible/faisable du problème en nombres entiers de maximisation fournit une borne inférieure sur la valeur de la solution optimale de ce problème en nombres entiers. C'est une borne supérieur sur un problème de minimisation.

D'une façon globale toute méthode de résolution qui garantit ces propriétés peut servir comme relaxation.

3.3.2 Approche par énumération

Un modèle en nombres entiers borné (par exemple, un modèle avec uniquement des variables 0–1) possède un nombre fini de solutions. Nous pourrions **envisager de toutes les énumérer**, mais le nombre de solutions explose rapidement. Pour $n = 20$ variables 0–1, il y a **plus d'un million de solutions possibles**. **Pour $n = 30$, c'est plus d'un milliard**. Comme il apparaît qu'il est vite déraisonnable de procéder à une énumération complète des solutions, nous allons essayer de mettre à profit la relaxation en programme linéaire pour éliminer certaines de ces solutions. Cette technique d'énumération partielle est connue sous le vocable de *branch-and-bound* **(B&B)**. Il s'agit d'une **approche** *diviser-pour-régner* :

— décomposition du problème en sous-problèmes plus simples ;
— combinaison de la résolution de ces sous-problèmes pour obtenir la solution du problème original.

Dans l'algorithme de branch-and-bound, chaque sous-problème correspond à un sommet dans l'arbre des solutions. Nous résolvons la relaxation linéaire de chaque sous-problème. L'information tirée de la relaxation linéaire nous permettra (peut-être) d'éliminer toutes les solutions pouvant être obtenues à partir de ce sommet.

3.3.2.1 Algorithme de branch & bound : cas 0–1

Un algorithme simple pour énumérer toutes les solutions d'un modèle 0–1 consiste à :

- choisir un sommet dans l'arbre des solutions ;
- choisir une variable x non encore fixée relativement à ce sommet ;
- générer les deux variables $x = 0$ et $x = 1$ (la variable x est dite *fixée*) : chaque alternative correspond à un sommet de l'arbre des solutions ;
- recommencer à partir d'un sommet pour lequel certaines variables ne sont pas encore fixées.

A la racine de l'arbre, aucune variable n'est encore fixée, tandis qu'aux feuilles de l'arbre, toutes les variables ont été fixées. Le nombre de feuilles est 2^n (pour n variables 0–1).

Le calcul de borne (ou évaluation) consiste à résoudre la relaxation linéaire en chaque sommet. L'élagage (ou élimination) consiste à utiliser l'information tirée de la résolution de la relaxation linéaire pour éliminer toutes les solutions émanant du sommet courant. Dés lors, le branch-and-bound est un algorithme de séparation et d'évaluation successives.

Exemple 12 (California Mfg)**.** *Rappelons le problème*

$$\begin{aligned} max\ & z = 9x_1 + 5x_2 + 6x_3 + 4x_4 \\ s.c.\ & 6x_1 + 3x_2 + 5x_3 + 2x_4 \leq 10; \\ & x_3 + x_4 \leq 1; \\ & -x_1 + x_3 \leq 0; \\ & -x_2 + x_4 \leq 0; \\ & x_1, x_2, x_3, x_4 \leq 1; \\ & x_1, x_2, x_3, x_4 \geq 0; \\ & x_1, x_2, x_3, x_4 \ \textit{entiers.} \end{aligned}$$

La relaxation linéaire permet aux variables de prendre les valeurs fractionnaires entre 0 et 1, ce qui conduit à la solution

$$\left(\frac{5}{6}, 1, 0, 1\right),$$

avec comme valeur $z = 16.5$*. Branchons sur la variable* x_1*.*

Dénotons sous-problème 1 celui obtenu avec $x_1 = 0$ *:*

$$\begin{aligned} max\ z &= 5x_2 + 6x_3 + 4x_4 \\ s.c.\ 3x_2 + 5x_3 + 2x_4 &\leq 10; \\ x_3 + x_4 &\leq 1; \\ x_3 &\leq 0; \\ -x_2 + x_4 &\leq 0; \\ x_2, x_3, x_4 &\ binaires. \end{aligned}$$

La solution de la relaxation linéaire est $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$*, avec* $z = 9$*.*

Le sous-problème 2 est celui obtenu avec $x_1 = 1$ *:*

$$\begin{aligned} max\ z &= 5x_2 + 6x_3 + 4x_4 + 9 \\ s.c.\ 3x_2 + 5x_3 + 2x_4 &\leq 4; \\ x_3 + x_4 &\leq 1; \\ x_3 &\leq 1; \\ -x_2 + x_4 &\leq 0; \\ x_2, x_3, x_4 &\ binaires. \end{aligned}$$

La solution de la relaxation linéaire est alors

$$(x_1, x_2, x_3, x_4) = \left(1, \frac{4}{5}, 0, \frac{4}{5}\right),$$

avec $z = 16 + \frac{1}{5}$*.*

Nous obtenons dès lors les bornes suivantes :
— sous-problème 1 : $Z_1 \leq 9$;
— sous-problème 2 : $Z_2 \leq 16 + \frac{1}{5}$.
Notons que toutes les variables sont binaires et tous les paramètres dans l'objectif sont des valeurs entières. Dès lors, la borne supérieure pour le sous-problème 2 est 16. Pour le sous-problème 1, la solution obtenue est entiere : c'est la meilleure solution courante. Nous savons que la valeur optimale cherchée, Z, sera au moins

$$Z^* = 9 : Z \geq Z^*.$$

Quels sous-problèmes pouvons-nous à présent considérer afin de nous approcher de la solution optimale ? Tous les sous-problèmes actuellement traités doivent-ils donner naissance à d'autres problèmes. Si un sous-problème ne donne lieu à aucun autre problème, nous parlerons d'élagage, en référence avec l'idée de couper la branche correspondante dans l'arbre d'exploration.

Considérons tout d'abord le sous-probleme 1 : la solution optimale de la relaxation PL est entière. Il ne sert donc à rien de brancher sur les autres variables, puisque toutes les autres solutions entières (avec $x_1 = 0$) sont nécessairement de valeur inférieures ou égales à 9. Nous pouvons donc élaguer ce sommet.

Pour le sous-probleme 2, la solution optimale de la relaxation PL n'est pas entière :

$$Z^* = 9 \leq Z \leq 16.$$

La branche ($x_1 = 1$) peut encore contenir une solution optimale. Mais si nous avions eu $Z_2 \leq Z*$, nous aurions pu conclure que la branche ne pouvait améliorer la meilleure solution courante.

Un sous-problème est elagué si une des trois conditions suivantes est satisfaite :

— test 1 : sa borne supérieure (valeur optimale de la relaxation PL) est inférieure ou égale à Z^* (valeur de la meilleure solution courante) ;

— test 2 : sa relaxation PL n'a pas de solution réalisable ;

— test 3 : la solution optimale de sa relaxation PL est entière.

Lorsque le test 3 est verifié, nous testons si la valeur optimale de la relaxation PL du sous-problème, Z_i, est supérieure à Z^*. Si $Z_i > Z^*$, alors $Z^* := Z_i$, et nous conservons la solution, qui devient la meilleure solution courante. En résumé, nous obtenons l'algorithme ci-dessous.

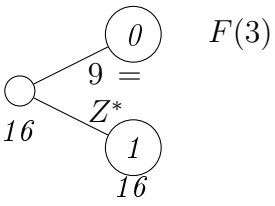
Algorithm 1. *Branch-and-Bound : cas binaire*

1. *Initialisation :*
   (a) *poser $Z^* = -\infty$;*
   (b) *appliquer le calcul de borne et les critères d'élagage à la racine (aucune variable fixée).*
2. *Critere d'arrêt : s'il n'y a plus de sous-problemes non élagués, arrêter.*
3. *Branchement :*
   (a) *parmi les sous-problèmes non encore élagués, choisir celui qui a été crée le plus récemment (s'il y a égalité, choisir celui de plus grande borne supérieure) ;*
   (b) *appliquer le Test 1 : si le sous-problème est élagué, retourner en 2.*
   (c) *brancher sur la prochaine variable non fixée.*
4. *Calcul de borne :*
   (a) *résoudre la relaxation PL de chaque sous-problème ;*
   (b) *arrondir la valeur optimale si tous les paramètres de l'objectif sont entiers.*
5. *Elagage : élaguer un sous-problème si*
   (a) *la borne superieure est inférieure ou égale à Z^* ;*
   (b) *la relaxation PL n'a pas de solution réalisable ;*
   (c) *la solution optimale de la relaxation PL est entière : si la borne supérieure est strictement supérieure à Z^*, Z^* est mise à jour et la solution de la relaxation PL devient la meilleure solution courante.*
6. *Retourner en 2.*

A partir de quel noeud devrions-nous brancher ? Il y a plusieurs choix possibles ; dans cette version, on propose comme règle de sélection de choisir le sous-problème le plus récemment créé. L'avantage est que cette approche facilite la réoptimisation lors du calcul de borne, car il n'y a que peu de changements apportés par rapport au dernier sous-probleme traité. Le désavantage est que cela peut créer un grand nombre de sous-probèmes. Une autre option est la règle de la meilleure borne : choisir le sous-problème ayant la plus grande borne supérieure.

Dans cette version, la règle de branchement consiste à choisir la prochaine variable non fixée. Il est souvent plus intéressant de choisir une variable à valeur fractionnaire. En branchant sur une telle variable, il est certain que les deux sous-problèmes crées mènent à des solutions differentes de la solution courante. De nombreux critères existent pour choisir une telle variable de facon à orienter la recherche vers un élagage rapide.

Exemple 13 (suite)**.** *Jusqu'à maintenant, voici l'arbre obtenu, en branchant sur la variable* x_1*.*



$F(3)$ *indique que le sous-probleme a été élagué (*fathomed*) en raison du Test 3.*

Sélection : nous choisissons le sous-problème 2, le seul qui n'a pas encore été élagué. Nous branchons sur la prochaine variable, soit x_2*. Deux nouveaux sous-problèmes sont crées :*

— *sous-problème 3 :* $x_1 = 1,\ x_2 = 0$ *;*

— *sous-problème 4 :* $x_1 = 1,\ x_2 = 1$*.*

Considérons tout d'abord le sous-problème 3 ($x_1 = 1,\ x_2 = 0$*). Nous obtenons le problème*

$$\begin{aligned} \max z_3 = {} & 6x_3 + 4x_4 + 9 \\ s.c.\ & 5x_3 + 2x_4 \leq 4 \\ & x_3 + x_4 \leq 1 \\ & x_3 \leq 1 \\ & x_4 \leq 0 \\ & x_3,\ x_4 \text{ binaire.} \end{aligned}$$

La solution de la relaxation PL est

$$(x_1, x_2, x_3, x_4) = \left(1, 0, \frac{4}{5}, 0\right).$$

et

$$Z = 13 + \frac{4}{5} :\ Z_3 \leq 13.$$

Le sous-problème 4 ($x_1 = 1,\ x_2 = 1$*) devient*

$$\begin{aligned} \max z_4 = {} & 6x_3 + 4x_4 + 14 \\ s.c.\ & 5x_3 + 2x_4 \leq 1 \\ & x_3 + x_4 \leq 1 \\ & x_3 \leq 1 \\ & x_4 \leq 1 \\ & x_3,\ x_4 \text{ binaire.} \end{aligned}$$

La solution de la relaxation PL est

$$(x_1, x_2, x_3, x_4) = \left(1, 1, 0, \frac{1}{2}\right).$$

et

$$Z = 16 : \ Z_4 \leq 16.$$

Aucun des tests d'élagage ne s'applique sur ces sous-problèmes. Nous devons dès lors choisir un des deux sous-problèmes pour effectuer un branchement, puisque ce sont ceux crées le plus récemment. Nous choisissons celui de plus grande borne supérieure, soit le sous-problème 4. Nous branchons sur x_3 et nous générons deux nouveaux sous-problèmes.

Le sous-problème 5, défini avec $x_1 = 1$, $x_2 = 1$, $x_4 = 0$, s'écrit La solution de la relaxation PL est

$$(x_1, x_2, x_3, x_4) = (1, 1, 0.2, 0)$$

et

$$Z = 14.$$

Le sous-problème 6, défini avec $x_1 = 1$, $x_2 = 1$, $x_4 = 1$, s'écrit La relaxation PL n'a pas de solution realisable : ce sous-probleme est élagué.

Le sous-problème 5 ne peut pas être élagué. Il est crée le plus récemment parmi les sous-problèmes non élagués (3 et 5), aussi choisissons-nous le pour effectuer un branchement. Nous branchons sur x_3 et générons les sous-problèmes suivants :

- *sous-problème 7 : $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, $x_4 = 0$;*
- *sous-problème 8 : $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$.*

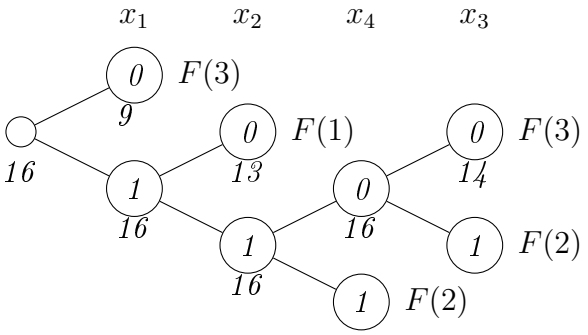
Toutes les variables sont fixées, aussi pouvons-nous directement résoudre ces sous-problèmes. Le sous-problème 7 a pour solution $(x_1, x_2, x_3, x_4) = (1, 1, 0, 0)$, pour $Z_7 = 14$. La solution est entière, aussi nous élaguons le sous-problème en vertu du Test 3. Puisque $Z_7 > Z^*$, $Z^* = 14$ et la solution du sous-problème devient la meilleure solution courante. Le sous-problème 8 a pour solution $(x_1, x_2, x_3, x_4) = (1, 1, 1, 0)$. Cette solution n'est pas réalisable. Le sous-problème est par conséquent élagué par le Test 2.

Le sous-problème 3 est le seul non encore élagué. Appliquons le Test 1 : $Z_3 = 13 \leq 14 = Z*$. Le sous-problème est donc élagué. Comme il n'y a plus de sous-problèmes non élagués, nous pouvons nous arrêter. La solution optimale est :

$$(x_1, x_2, x_3, x_4) = (1, 1, 0, 0),$$

et la valeur optimale est $Z^* = 14$.

L'arbre obtenu suite à l'exécution de l'algorithme se présente comme suit :



$F(j)$: le sous-problème est élagué par le Test j

3.3.2.2 Algorithme de branch & bound : cas général

Considérons à présent le cas général d'un modèle de programmation (mixte) en nombres entiers : variables entières générales et variables continues. Comme précédemment, nous ignorons dans un premier temps les contraintes d'intégralité (les valeurs des variables entières sont traitées comme variables continues), et résolvons le programme linéaire résultant. Si la solution optimale de ce programme satisfait aux contraintes d'intégralité, alors cette solution est aussi solution optimale du programme avec variables entières. Sinon, il doit exister au moins une variable x_j dont la valeur α est fractionnaire. La procédure de branchement se généralise alors comme suit : nous séparons alors le problème relaxé en deux sous-problèmes ; un sous-problème contiendra la contrainte $x_j \leq \lfloor \alpha \rfloor$ et le second la contrainte $x_j \geq \lceil \alpha \rceil = \lfloor \alpha \rfloor + 1$. Nous répétons le processus pour chacun des sous-problèmes. Cette procédure est habituellement représentée sous forme d'un arbre binaire où, à chaque niveau, une partition du sommet père s'effectue suivant la règle décrite précédemment. Il s'agit alors de parcourir cet arbre d'énumération afin d'y trouver la solution optimale. L'exploration d'un chemin de l'arbre peut prendre fin pour trois raisons :

- la solution devient entière ;
- le domaine admissible d'un sous-problème devient vide ;
- la valeur de l'objectif correspondant à la solution optimale du problème relaxé est inférieure (moins bonne) à celle d'une solution admissible connue, possiblement obtenue à un autre sommet de l'arbre.

Dans chacun de ces trois cas on dit que le sommet est sondé, et il est inutile de pousser plus loin dans cette direction. L'algorithme s'arrête lorsque tous les sommets sont sondés. La meilleure solution obtenue au cours du déroulement de l'algorithme est alors l'optimum global de notre problème.

Algorithm 2. *Algorithme de B&B : cas général*

1. *Initialisation :*
   (a) *Poser $Z^* = -\infty$.*
   (b) *Appliquer le calcul de borne et les critères d'élagage à la racine (aucune variable fixée).*
   (c) *Critere d'arrêt : s'il n'y a plus de sous-problèmes non élagués, arrêter.*
2. *Branchement :*
   (a) *Parmi les sous-problèmes non encore élagués, choisir celui qui a été crée le plus récemment (s'il y a égalité, choisir celui de plus grande borne supérieure).*
   (b) *Appliquer le Test 1 : si le sous-problème est élagué, retourner en 2.*
   (c) *Brancher sur la prochaine variable entière à valeur non entière dans la relaxation PL.*
3. *Calcul de borne : résoudre la relaxation PL de chaque sous-problème.*
4. *Elagage : élaguer un sous-problème si*

(a) La borne superieure est inférieure ou égale à Z^*.

(b) La relaxation PL n'a pas de solution réalisable.

(c) Dans la solution optimale de la relaxation PL, toutes les variables entières sont à valeurs entières : si la borne supérieure est strictement supérieure à Z^*, Z^* est mise à jour et la solution de la relaxation PL devient la meilleure solution courante.

5. Retourner en 2.

Un sous-problème est elagué si une des trois conditions suivantes est satisfaite :
— test 1 : sa borne supérieure (valeur optimale de la relaxation PL) est inférieure ou égale à Z^* (valeur de la meilleure solution courante) ;
— test 2 : sa relaxation PL n'a pas de solution réalisable ;
— test 3 : la solution optimale de sa relaxation PL est entière.

3.4 Branch and bound : exemple

Un programme linéaire mixte (MIP) se présente comme suit :

$$(MIP) \equiv \begin{cases} \text{minimize} & cx \\ \text{subject to} & Ax \leq b \\ & x_i \in \mathbb{Z}, i = 1..p \\ & x_i \in \mathbb{R}, i = p+1..n \end{cases} \tag{3.1}$$

Le problème : Comment le résoudre correctement et rapidement ?

Branch and Bound (BB) Diviser pour régner. "Diviser/Branch" : subdivise le problème. "Régner/Bound" : considérer la qualité de la solution des sous-problèmes.

Branch and Cut (BC) BB en renforçant la relaxation linéaire LP d'un MIP avec des inégalités avant tout branchement. On raisonne sur les contraintes !

Branch and Price (BP) BB en se concentrant sur le choix de la variable entrante (column generation/pricing) dans la résolution de la relaxation. On raisonne sur les variables (les colonnes) !

Branch and bound
— Diviser le problème en sous-problèmes
— Calculer la relaxation LPR du sous-problème en considérant "réelles" les variables entières
  — LPR infaisable : stop.
  — LPR a une solution faisable entière : résolu, solution optimale pour le sous-problème.
  — LPR a une solution moins bonne que la meilleure solution entière : stop.
  — Sinon LPR a des composantes réelles, Diviser en sous-problèmes.

Soit le problème MIP [1] :

1. http ://www.ie.bilkent.edu.tr/ mustafap/courses/bb.pdf

$$\begin{array}{ll} \text{maximize} & z = -x_1 + 4x_2 \\ \text{subject to} & -10x_1 + 20x_2 \leq 22 \\ & 5x_1 + 10x_2 \leq 49 \\ & x_1 \leq 5 \\ & x_i \geq 0, x_1 \text{ et } x_2 \text{ sont entiers.} \end{array} \tag{3.2}$$

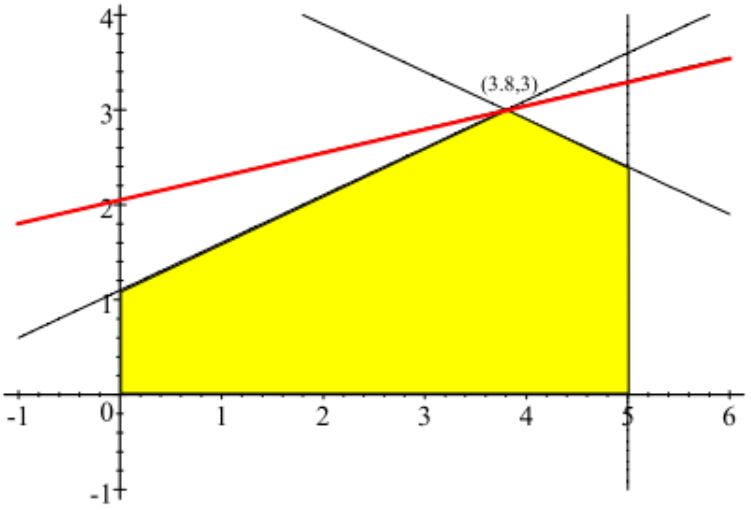
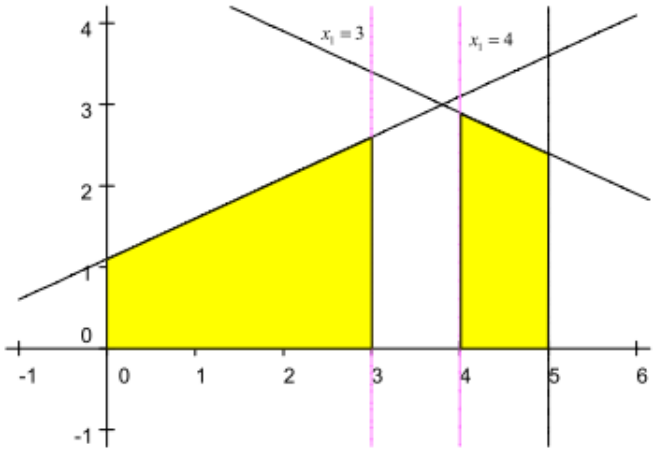


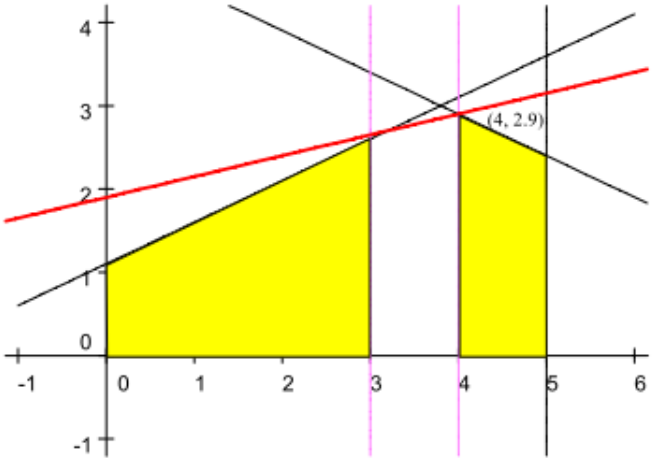
FIGURE 3.3 – Relaxation linéaire (3.3)

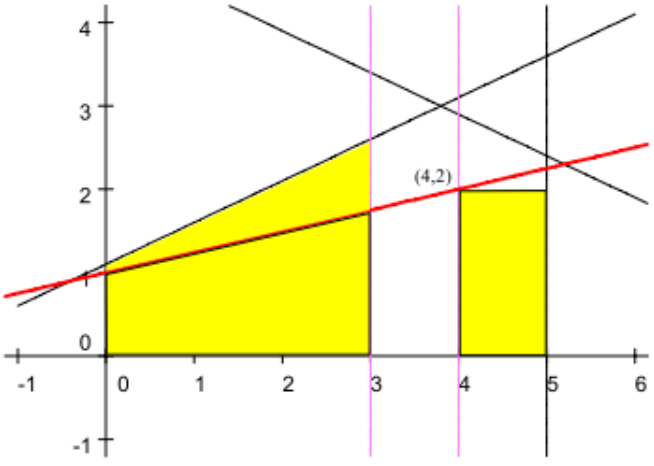
Au niveau de la relaxation

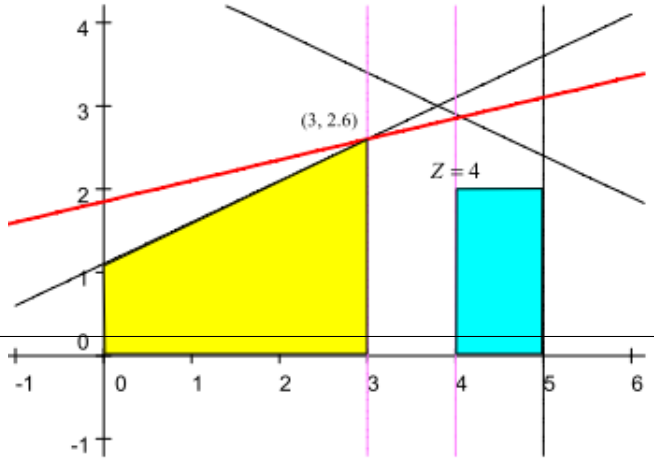
$$\begin{array}{ll} \text{maximize} & z = -x_1 + 4x_2 \\ \text{subject to} & -10x_1 + 20x_2 \leq 22 \\ & 5x_1 + 10x_2 \leq 49 \\ & x_1 \leq 5 \\ & x_i \geq 0, x_1 \text{ et } x_2 \text{ sont réelles.} \end{array} \tag{3.3}$$

La relaxation (3.3) est illustrée dans la Figure 3.3. **Sa solution est** $(3.8, 3)$ **avec** $z = 8.2$**.** Nous devons faire un "Branch" sur la variable fractionnaire x_1, en obtenant deux branches $x_1 \geq 4$ et $x_1 \leq 3$. Cette séparation (Branch) est illustrée dans la première image de la Figure 3.4.


$x_1 = 3$
$x_1 = 4$


(4, 2.9)


(4,2)


(3, 2.6)
Z = 4

Par la suite la procédure Branch and Bound procède comme suit :

1. Le sous-problème correspondant à $x_1 \geq 4$ est résolu sur $\mathbb{R}$. On obtient la solution $(4, 2.9)$ avec $z = 7.6$. On obtient ainsi deux sous-sous-problèmes $x_2 \geq 3$ et $x_2 \leq 2$.

2. Le sous-problème $x_2 \geq 3$ est infaisable.

3. Nous considérons donc maintenant le sous-problème $x_2 \leq 2$. Ce dernier a une solution optimale (donnée dans la deuxième image de la Figure 3.4, qui a comme solution $(4, 2)$ avec $z = 4$. Cette première solution va permettre d'avoir une meilleure borne (inférieure) $z^* = 4$ de notre problème avec une valeur $z = 4$.

4. En ce qui concerne la branche $x_1 \leq 3$, on obtient la solution $(3, 2.6)$ avec $z = 7.4$, donnant lieu à deux branches $x_2 \leq 2$ et $x_2 \geq 3$.

5. Le LPR de $x_2 \geq 3$ est infaisable.

6. En ce qui concerne $x_2 \leq 2$, on obtient la solution $(1.8, 2)$ avec $z = 6.2$, donnant lieu à deux branches $x_1 \geq 2$ et $x_1 \leq 1$.

7. Pour $x_1 \geq 2$, on obtient la solution $(2, 2)$ avec $z = 6$. La solution est entière, d'où la nouvelle borne $z^* = 6$.

8. Pour $x_1 \leq 1$, nous obtenons une solution $(1, 1.6)$ avec $z = 5.4$. 5.4 est inférieur à la meilleure borne courante z^*, d'où son élagage.

Soit un autre problème :

$$\begin{array}{ll} \text{maximize} & z = 9 + 5x_2 + 6x_3 + 4x_4 \\ \text{subject to} & 3x_2 + 5x_3 + 2x_4 \leq 4 \\ & x_3 + x_4 \leq 1 \\ & -x_1 + x_3 \leq 0 \\ & x_2 + x_4 \leq 0 \\ & x_i \in \{0, 1\} \end{array} \tag{3.4}$$

La procédure Branch and Bound est illustré dans la Figure 3.5

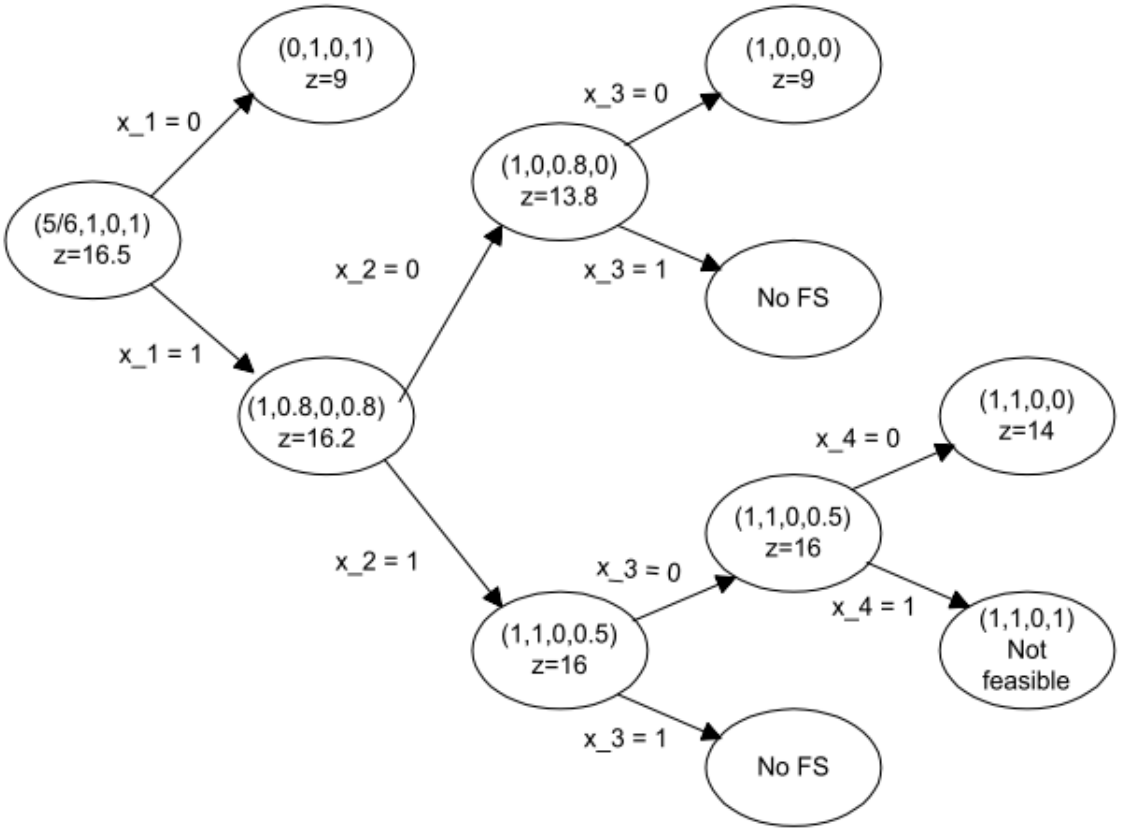


FIGURE 3.5 – Arbre de recherche de la procédure "Branche and Bound" sur le problème (3.3)

3.5 Travaux Dirigés I (Modélisation)

3.5.1 Exercice

Une entreprise de production de produits alimentaires désire orienter ses activités sur 3 lignes de produits I, II et III. Le profit moyen par produit est estimé à 300 DA par tonne pour le produit I, 200 DA pour le produit II, et 500 DA pour le produit III. Les équipements sont répartis en trois départements de production :

— Fabrication ;
— Mélange ;
— Emballage.

La durée maximale de charge pour chaque département est de 8 heures par jour. Les processus de production relatifs aux trois produits font l'objet des opérations successives suivantes :

— Produit I : Fabrication-Mélange. La production est enlevée par les utilisateurs dès qu'elle est réalisée. Chaque tonne ainsi produite exige 3 heures d'utilisation de la capacité de "fabrication" et 1 heures de capacité du département "mélange".
— Produit II : Mélange-Emballage. Le produit est réalisé à partir d'achats de composantes alimentaires non produites dans l'entreprise, fait l'objet des seules opérations de mélange et emballage. Chaque tonne produite exige 1 heure d'utilisation de capacité "mélange" et 2 heures de capacité "emballage".
— Produit III : Fabrication-Mélange-Emballage. Le produit subit les 3 séries d'opérations fabrication-mélange-emballage, chacune d'entre elles exigeant respectivement 2 heures, 1 heure et 1 heure de capacité des équipements.

1. Ecrire le programme linéaire modélisant ce problème ?

3.5.2 Exercice

Dans une entreprise de construction de matériel électrique, on dispose de 1200 heures/machine par mois et 1500 heures/ouvrier par mois. Les contacteurs nécessitent 15 heures/machine, 12 heures/ouvrier et rapportent 8 DA (par unité). Les disjoncteurs nécessitent 30 heures/machine, 30 heures/ouvrier et rapportent 20 DA (par unité). Les compteurs nécessitent 20 heures/machine, 25 heures/ouvrier et rapportent 18 DA (par unité).

1. Modéliser sous forme d'un programme linéaire PL la recherche des quantités optimales des contacteurs x_1, des disjoncteurs x_2 et des compteurs x_3 pour maximiser le gain ?
2. Formuler le dual du PL ?
3. Démontrer que la contrainte $17x_1 + 60x_2 + 45x_3 \leq 2700$ est non-redondante [2] dans le PL ?
4. Si des contraintes sont redondantes dans un PL, comment se comporterait l'algorithme du simplex ? Comment y remédier ?

2. Soit le système d'inéquations $l_i \equiv \sum_{j=1\ldots m} a_{i,j}x_j \geq c_i$, avec $i = 1\ldots n$. Une inéquation $l' \equiv \sum_{j=1\ldots m} b_j x_j \geq d$ est dite redondante si et seulement si $l' = \sum_{i=1\ldots n} \lambda_i l_i$ où $\forall i, \lambda_i$ est un réel positif.

3.5.3 Exercice

Etant donné deux variables x_1, x_2 binaires (i.e., $x_1 \in \{0,1\}$, $x_2 \in \{0,1\}$). Soit l'expression non linéaire de leur multiplication :

$$(P) \begin{cases} y = x_1 \times x_2 \\ x_1 \in \{0,1\}, x_2 \in \{0,1\}, y \in \{0,1\} \end{cases} \tag{3.5}$$

Nous proposons un système d'équations linéaires (Q)

$$(Q) \begin{cases} y \leq x_1 \\ y \leq x_2 \\ y \geq x_1 + x_2 - 1 \\ x_1 \in \{0,1\}, x_2 \in \{0,1\}, y \in \{0,1\} \end{cases} \tag{3.6}$$

Démontrer que (P) est équivalent à (Q) ?

3.5.4 Exercice

Soit le problème

$$(P) \begin{cases} \min \sum_{i=1..n} c_i x_i \\ \\ (\sum_{j=1..n} A_j x_j \leq b) \text{ OU } (\sum_{j=1..n} C_j x_j \leq d) \\ x_i \geq 0, i = 1..n \end{cases} \tag{3.7}$$

Le problème (P) n'est pas un PL car il contient la contrainte de disjonction "OU". Nous proposons un autre problème (Q)

$$(Q) \begin{cases} \min \sum_{i=1..n} c_i x_i \\ \sum_{j=1..n} A_j x_j \leq b + My \\ \sum_{j=1..n} C_j x_j \leq d + M(1-y) \\ x_i \geq 0, i = 1..n \\ y \in \{0,1\} \end{cases} \tag{3.8}$$

où M est un nombre très grand.

Démontrer que (P) est équivalent à (Q) ?

3.5.5 Exercice

Une firme *Penault* produit deux types de carrosseries C_1 et C_2. Nous nous intéressons à deux tâches indépendantes :

- *Peinture* : Cette firme ne pourra pas peindre C_2 à partir d'un maximum de 40 carrosseries C_1. Cette firme ne pourra pas aussi peindre C_1 à partir d'un maximum de 60 carrosseries C_2.
- *Fabrication* : Cette firme ne pourra pas produire C_2 à partir d'un maximum de 50 carrosseries C_1. Cette firme ne pourra pas aussi fabriquer C_1 à partir d'un maximum de 50 carrosseries C_2.

La carrosserie C_1 rapporte 300 DA et C_2 200 DA.

1. Modéliser sous forme d'un programme linéaire PL la recherche des quantités optimales x_1 de C_1 et x_2 de C_2 à produire pour maximiser le gain ? (N.B. Dans une tâche T donnée dans la firme, si la firme ne pourra pas produire C_2 à partir de k_1 C_1, et la firme ne pourra pas aussi produire C_1 à partir de k_2 C_2, la contrainte T peut être traduite en une seule contrainte linéaire $(x_1/k_1) + (x_2/k_2) \leq 1$.)
2. Tracer graphiquement le polyèdre décrivant le PL et extraire la solution optimale sans passer par l'algorithme du Simplexe ?
3. Soit la contrainte supplémentaire suivante : la firme *Penault* doit produire au moins 30 carrosseries C_1 et au moins 20 carrosseries C_2. Comment devient-il le PL ?

3.6 Travaux Dirigés II (séparation/évaluation)

3.6.1 Exercice [Problème d'affectation]

Soit un problème d'affectation ayant comme données :
— n tâches à affecter à n personnes
— Toute personne exécute 1 et 1 seule tâche.
— $c_{i,j}$ est le coût d'affectation de la tâche i à la personne j.
Le problème : trouver l'affectation des tâches aux personnes qui minimise le coût total ?
Soit l'instance du problème :

	Tâche 1	Tâche 2	Tâche 3	Tâche 4
Personne a	9	2	7	8
Personne b	6	4	3	7
Personne c	5	8	1	8
Personne d	7	6	9	4

Soit le calcul suivant :

$$LB1 = \sum_{i=1..n} \min_{j=1..m} c_{i,j}$$

Questions :

1. Démontrer que $LB1$ est une méthode de relaxation valide.
2. Dérouler l'algorithme par séparation évaluation sur l'instance donnée en exploitant $LB1$ comme relaxation.

3.6.2 Exercice [Problème de sac-à-dos]

Soit un problème de sac-à-dos ayant comme données :
— Un sac de volume V.
— Un ensemble de n objects $O = \{o_1, o_2, ..., o_n\}$.
— Tout objet o_i a un volume v_i et une utilité u_i.
— On suppose que $\sum_{i=1..n} v_i > V$.
Le problème : Trouver un sous-ensemble d'objets de O qui maximise l'utilité et qui soit borné par V.
Soit le calcul LB2 suivant :
— Soit le vecteur R des rapports entre les utilités et les volumes $[u_1/v_1, u_2/v_2, ..., u_n/v_n]$.
— Soit $x = (x_1, x_2, ..., x_n)$ les variables binaires associées aux n objets. $x_i = 0$ si l'objet i est mis dans le sac, sinon 1.
— Le calcul consiste en :
  — Trier R par ordre décroissant ;
  — $Disponible = V$;
  — Initialiser x à zéro ;
  — Pour $i = 1$ jusqu'à n faire :
    — Si ($v_i \leq Disponible$) alors $x_i = 1$; sinon $x_i = Disponible/v_i$ finsi.
    — $Disponible = Disponible - v_i * x_i$;

Questions :

1. Démontrer que $LB2$ est une méthode de relaxation valide.

2. Dérouler l'algorithme par séparation évaluation sur l'instance donnée en exploitant $LB2$ comme relaxation sur les données suivantes :

Objets	1	2	3	4
u	16	22	12	8
v	5	7	4	3
V	14			

3. Soit maintenant un autre algorithme UB2 qui reprend l'algorithme LB2 mais en trouvant une solution faisable (sans garantie d'optimalité) dans une démarche gloutonne :
   — Soit le vecteur R des rapports entre les utilités et les volumes $[u_1/v_1, u_2/v_2, ..., u_n/v_n]$.
   — Soit $x = (x_1, x_2, ..., x_n)$ les variables binaires associées aux n objets. $x_i = 0$ si l'objet i est mis dans le sac, sinon 1.
   — Le calcul consiste en :
      — Trier R par ordre décroissant ;
      — $Disponible = V$;
      — Initialiser x à zéro ;
      — Pour $i = 1$ jusqu'à n faire :
         — Si $(v_i \leq Disponible)$ alors $x_i = 1$; sinon $x_i = 0$ finsi.
         — $Disponible = Disponible - v_i * x_i$;
4. Dérouler à nouveau l'algorithme par séparation évaluation sur l'instance donnée en exploitant $LB2$ et $UB2$ sur les données ci-dessus.
5. Formuler ce problème en terme d'un programme linéaire en nombres entiers ?
6. Résoudre l'instance du problème par séparation/évaluation en utilisant la relaxation linéaire. Faites vous aider avec le solveur ampl pour le calcul de borne via l'algorithme du simplexe. Comparer l'arbre de recherche utilisant la relaxation linéaire avec l'arbre via la borne $LB2$?

3.7 Travaux Pratiques (Optimisation)

3.7.1 Exercice [Un modèle simple]

Installer `AMPL` en décompressant "amplide-demo-linux32.tar.gz", la version "AMPL FOR STUDENTS" à partir du site officiel http://ampl.com, puis invoquer AMPL :

```
tar xzf amplide-demo-linux32.tar.gz
ample
```

Commençons par un exemple simple d'un problème d'optimisation.

Une compagnie de commercialisation de la peinture fournit deux types de couleurs : la couleur bleue et la couleur dorée. La couleur bleue est vendu 10 DA par gallon, et la dorée 15 DA. La compagnie a une unité de fabrication et fabrique une couleur par unité de temps. Cependant, la couleur bleue est plus facile à produire et l'unité de fabrication peut produire 40 gallons par heure, et 30 gallons pour la couleur dorée. L'unité de vente de cette compagnie informe qu'elle ne peut pas vendre plus de 860 gallons de la couleur dorée et 1000 gallons de la couleur bleue. Supposons que dans le volume horaire travaillé par semaine est de 40 heures, et le produit est toujours stocké dans la semaine qui suit. Nous voulons déterminer le nombre de gallons de couleur dorée et bleue à produire pour maximiser le gain.

Soit `PaintB` (resp. `PaintG`) le nombre de gallons à produire de la peinture de couleur bleue (resp. de couleur dorée). Le problème peut être formulé comme suit :

$$\begin{array}{ll} \max & 10PaintB + 15PaintG \\ s.t. & \frac{1}{40}PaintB + \frac{1}{30}PaintG \leq 40 \\ & 0 \leq PaintB \leq 1000 \\ & 0 \leq PaintG \leq 860 \end{array}$$

`AMPL` permet de concevoir des programmes mathématiques à saisir avec une syntaxe qui est très proche de la notation algébriques que l'on vient de voir. Pour utiliser `AMPL`, on doit créer un fichier texte qui va contenir le texte du programme mathématique (On utilisera un éditeur de texte).

Saisir le texte suivant dans l'éditeur :

```
## Example One
var PaintB; # amount of blue
var PaintG; # amount of gold

maximize profit: 10*PaintB + 15*PaintG;

subject to time: (1/40)*PaintB + (1/30)*PaintG <= 40;
subject to blue_limit: 0 <= PaintB <= 1000;
subject to gold_limit: 0 <= PaintG <= 860;
```

Sauvegarder l'exemple sous le nom `examp1.mod`.

Notez les différents opérateurs et expressions utilisés pour déclarer les variables, la fonction objectif et les contraintes.

Suivre les étapes suivantes pour manipuler notre exemple :

— `AMPL` est un environnement de programmation mathématique qui peut s'interfacer avec la plupart des solveurs connus (`Cplex`, `Minos`, `lpsolve`, ...). Précisons maintenant à `AMPL` le solveur que l'on veut utiliser :
`ample: option solver cplex;`
— Charger notre exemple
`ample: model examp1.mod;`
— Si vous avez des erreurs au niveau de la saisie, vous pouvez revenir à votre éditeur de texte, corriger puis recharger le fichier, en commençons par vider l'environnement :
`ample: reset;`
— Pour résoudre le problème, il suffit d'invoquer
`ample: solve;`
— `AMPL` affiche
`CPLEX 8.0.0: optimal solution; objective 17433.33333`
`2 simplex iterations (0 in phase I)`
— Pour afficher la solution, il suffit d'invoquer la fonction d'affichage `display` avec les variables à afficher
`ample: display PaintB, PaintG;`
— Vous pouvez rediriger la sortie vers un fichier
`ample: display PaintB, PaintG > examp1.out`

3.7.2 Exercice [Un modèle plus général]

Le problème que l'on vient de manipuler est figé ; cependant un nombre de problèmes économiques d'optimisation peuvent se formuler dans le même type de modèle mais se portant sur un nombre de produits, des coûts, et des volumes horaires quelconques. Soit ici ce modèle général :

Soient	n : le nombre de couleurs
	t : le temps total disponible
	p_i : le profit par gallon i, $i = 1..n$
	r_i : gallons par heure de la couleur i, $i = 1..n$
	m_i : la demande maximale de la couleur i, $i = 1..n$
Variables	x_i : le nombre de gallons de la couleur i, $i = 1..n$
Maximiser	$\sum_{i=1..n} p_i x_i$
Subject to	$\sum_{i=1..n} (1/r_i) x_i \leq t$
	$0 \leq x_i \leq m_i$ pour tout i, $i = 1..n$.

Ainsi, notre modèle simple se résume en le modèle général ci-dessus où $n = 2, t = 40, p_1 = 10, p_2 = 15, r_1 = 40, r_2 = 30, m_1 = 1000$ et $m_2 = 860$.

Justement `AMPL` permet de saisir des modèle généraux permettant de les paramétrer aisément. Le modèle général est toujours stocké dans un fichier à part, soit par exemple `model.mod`. Le paramétrage ou la fixation des paramètres sur une instance donnée est faite dans un fichier séparé, dit fichier des données, soit *model.dat*.

Dans notre cas, nous pouvons saisir le modèle général de la compagnie dans un fichier `examp1g.mod` comme suit :

```
param n;
param t;
param p{i in 1..n};
param r{i in 1..n};
param m{i in 1..n};
var paint{i in 1..n};

maximize profit: sum{i in 1..n} p[i]*paint[i];

subject to time: sum{i in 1..n} (1/r[i])*paint[i] <= t;
subject to capacity{i in 1..n}: 0 <= paint[i] <= m[i];
```

Soit le fichier de données `examp1g.dat` qui contient le jeu de données pour notre premier modèle simple :

```
## Example Two - Data
param n:= 2;
param t:= 40;

param p:= 1 10
          2 15;
param r:= 1 40
          2 30;
param m:= 1 1000
          2 860;
```

Chargeons maintenant ces fichiers pour résoudre encore le modèle simple dans cette nouvelle modélisation générale :

```
ample: reset
ample: model examp1g.mod;
ample: data examp1g.dat;
ample: solve;
```

Invoquons l'affichage de la solution :

```
ample: display paint;
```

3.7.3 Exercice [Une autre manière pour saisir les données]

On peut saisir tous les paramètres en une seule fois, avec :

```
## Example Two - Another way to write the data
param n:= 2;
param t:= 40;
param: p r m:=
  1 10 40 1000
  2 15 30 860;
```

3.7.4 Exercice [Les ensembles]

Dans la tâche de la modélisation, nous avons souvent besoin de garder la nomenclature de l'environnement dans lequel le problème a été posé. Soit par exemple la formulation

```
## Example Two - Model with sets
set P;
param t;
param p{i in P};
param r{i in P};
param m{i in P};
var paint{i in P};

maximize profit: sum{i in P} p[i]*paint[i];

subject to time: sum{i in P} (1/r[i])*paint[i] <= t;
subject to capacity{i in P}: 0 <= paint[i] <= m[i];
```

Ainsi, on voit que l'ensemble des couleurs est maintenant un ensemble que l'on peut énumérer en utilisant des noms réels :

```
## Example Two - Data with sets
set P:= blue gold;
param t:= 40;
param p:= blue 10 gold 15;
param r:= blue 40 gold 30;
param m:= blue 1000 gold 860;
```

Et aussi

```
## Example Two - Data with sets
set P:= blue gold;
param t:= 40;
param: p r m:=
blue 10 40 1000
gold 15 30 860;
```

3.7.5 Exercice [Des paramètres et des variables de deux dimensions]

Nous avons souvent besoin de faire appel à des modèles où les paramètres et les variables ont plusieurs dimensions, et tout particulièrement deux dimensions.

Soit le problème suivant :

La compagnie a fait une extension, et a maintenant trois dépôts pour stocker la peinture bleue. Dans une semaine, la peinture doit être acheminer à différents clients. Pour chaque dépôt avec chaque client, les coûts d'acheminement sont différents. Nous donnons ci-dessous ces coûts.

	Cust 1	Cust. 2	Cust. 3	Cust. 4
Warehouse 1	1	2	1	3
Warehouse 2	3	5	1	4
Warehouse 3	2	2	2	2

Note de traduction : Warehouse est le dépôt, Customer est le client.

Le nombre de gallons disponibles au niveau de chaque dépôt, et le nombre de gallons demandé par chaque client sont donnés respectivement comme suit :

```
Warehouse 1 250
Warehouse 2 800
Warehouse 3 760

Customer 1: 300
Customer 2: 320
Customer 3: 800
Customer 4: 390
```

Le modèle `AMPL` de ce problème est donné comme suit :

```
## Example Three - Model
param warehouse; # number of warehouses
param customer; # number of customers

#transportation cost from warehouse i
#to customer j
param cost{i in 1..warehouse, j in 1..customer};
param supply{i in 1..warehouse}; #supply at warehouse i
param demand{i in 1..customer}; #demand at customer j

var amount{i in 1..warehouse, j in 1..customer};

minimize Cost: sum{i in 1..warehouse, j in 1..customer}
cost[i,j]*amount[i,j];

subject to Supply {i in 1..warehouse}: sum{j in 1..customer} amount[i,j] = supply[i];
subject to Demand {j in 1..customer}: sum{i in 1..warehouse} amount[i,j] = demand[j];
subject to positive{i in 1..warehouse, j in 1..customer}: amount[i,j]>=0;
```

Stockons ce modèle dans un fichier `examp2.mod`.

Et le jeu de données qui suit dans `examp2.dat`.

```
## Example Three - Data
param warehouse:= 3;
param customer:= 4;
param cost: 1 2 3 4 :=
1 1 2 1 3
```

```
2 3 5 1 4
3 2 2 2 2;

param supply:=
1 250
2 800
3 760;

param demand:=
1 300
2 320
3 800
4 390;
```

On peut aussi saisir les noms propres des différents paramètres pour obtenir un modèle qui reprend les mêmes termes que ceux utilisés en pratique.

```
## Example Three - Model using sets
set Warehouses;
set Customers;

#transportation cost from warehouse i
#to customer j
param cost{i in Warehouses, j in Customers};
param supply{i in Warehouses}; #supply at warehouse i
param demand{j in Customers}; #demand at customer j
var amount{i in Warehouses, j in Customers};

minimize Cost: sum{i in Warehouses, j in Customers} cost[i,j]*amount[i,j];

subject to Supply {i in Warehouses}: sum{j in Customers} amount[i,j]=supply[i];
subject to Demand {j in Customers}: sum{i in Warehouses} amount[i,j]=demand[j];
subject to positive{i in Warehouses, j in Customers}: amount[i,j]>=0;
```

Et le jeu de données comme suit :

```
## Example Three - Data with sets
set Warehouses := Oakland San_Jose Albany;
set Customers:= Home_Depot K_mart Wal_mart Ace;

param cost: Home_Depot K_mart Wal_mart Ace :=
Oakland   1 2 1 3
San_Jose  3 5 1 4
Albany    2 2 2 2;

param supply:=
Oakland  250
```

```
San_Jose 800
Albany   760;

param demand:=
Home_Depot 300
K_mart     320
Wal_mart   800
Ace        390;
```

3.7.6 Exercice [Programmation en nombres entiers]

Souvent, dans les modèles mathématiques, certaines variables peuvent être entières. AMPL facilité la manipulation de ces variables via les mots clés `integer` et `binary` que l'on peut placer après le nom de la variable pour dire que la variable en question est respectivement entière ou prend ses valeurs dans le domaine binaire $\{0, 1\}$.

Soit une modification au niveau des capacités des dépôts :

```
Warehouse 1 550
Warehouse 2 1100
Warehouse 3 1060
```

Cette augmentation vient avec un coût donné ci-dessous

```
Warehouse 1 500
Warehouse 2 500
Warehouse 3 500
```

Nous voyons ici que la demande excède la produits disponibles.

Soit maintenant le modèle suivant qui prend en compte ces nouvelles spécificités :

```
## Example Four - Mixed-IP model file for the warehouse location
  problem
set Warehouses;
set Customers;
#transportation cost from warehouse i
#to customer j
param cost{i in Warehouses, j in Customers};
param supply{i in Warehouses}; #supply capacity at warehouse i
param demand{j in Customers}; #demand at customer j
param fixed_charge{i in Warehouses}; #cost of opening warehouse j

var amount{i in Warehouses, j in Customers};
var open{i in Warehouses} binary; # = 1 if warehouse i is opened, 0 otherwise

minimize Cost: sum{i in Warehouses, j in Customers}
                 cost[i,j]*amount[i,j] + sum{i in Warehouses} fixed_charge[i]*open[i];
```

```
subject to Supply {i in Warehouses}: sum{j in Customers} amount[i,j] <= supply[i]
subject to Demand {j in Customers}: sum{i in Warehouses} amount[i,j]=demand[j];
subject to positive{i in Warehouses, j in Customers}: amount[i,j]>=0;
```

Soit le jeu de données suivant :

```
## Example Four - Data file for the warehouse location problem
set Warehouses:= Oakland San_Jose Albany;
set Customers:= Home_Depot K_mart Wal_mart Ace;

param cost: Home_Depot K_mart Wal_mart Ace:=
Oakland  1 2 1 3
San_Jose 3 5 1 4
Albany   2 2 2 2;

param supply:=
Oakland  550
San_Jose 1100
Albany   1060;

param demand:=
Home_Depot 300
K_mart     320
Wal_mart   800
Ace        390;

param fixed_charge:=
Oakland  500
San_Jose 500
Albany   500;
```

Faites les tests de résolution !

3.7.7 Exercice [Mise en pratique des modèles étudiés]

1. Mettre en forme le problème de l'exemple 2 sous forme AMPL. Commentez !
2. Mettre en forme le problème de l'exemple 3 sous forme AMPL, en proposant des paramètres significatifs. Commentez !
3. Soit le problème de localisation de l'exemple 3 : modifier le modèle en supposant qu'un site a plusieurs capacités de production à décider !
4. Mettre en forme le problème de l'exemple d'un objectif avec coûts fixes sous forme AMPL. Commentez !
5. Mettre en forme le problème de l'exemple 7 sous forme AMPL, en mettant M à une très grande valeur. Comment se comporte le modèle si M est petit ? Commentez !
6. Mettre en forme le problème de l'exemple 8 sous forme AMPL. Commentez !

7. Mettre en forme le problème de l'exemple 9 sous forme AMPL, dans sa forme générique, en séparant entre le modèle et les données des paramètres.

3.7.8 Exercice [Séparation/Evaluation]

1. Déroulez à nouveau la résolution du cas 0-1 de l'algorithme par séparation/évaluation en exploitant le solveur AMPL sur le domaine réel.
2. Déroulez à nouveau la résolution du cas $I\!N$de l'algorithme par séparation/évaluation en exploitant le solveur AMPL sur le domaine réel.

3.7.9 Exercice [Programmation nonlinéaire]

Assez souvent, on fait appel dans les modèles à des fonction nonlinéaires qui donnent lieu à des programmes nonlinéaires.

Première chose, il faut maintenant faire appel à un solveur qui sait manipuler ces problèmes. `Cplex` sait résoudre des programmes linéaires classiques et les programmes linéaire en nombres entiers ou mixte. Pour les modèles nonlinéaire, on fera appel au solveur `Minos`.

```
option solver minos;
```

Soit le problème suivant :

Supposons que nous avons plusieurs alternatives d'investissement A. Et nous savons évaluer le retour sur investissement pour certaines années T. A partir de ces données, on peut calculer la matrice des covariance des investissements.

On veut donc :

— *maximiser le retour sur investissement sous le maximum des risques potentiels.*

— *minimiser les risques sous le minimum de retour sur investissement.*

Le modèle suivant reprend les termes de ce problème d'une façon générale.

```
##Example 5 - Nonlinear Portfolio
set A; # asset categories
set T := {1984..1994}; # years
param s_max default 0.00305; # i.e., a 5.522 percent std. deviation on reward
param R {T,A};
param mean {j in A} := ( sum{i in T} R[i, j] - mean[j];
param Rtilde {i in T, j in A} := R[i,j] - mean[j];

var alloc{A} >=0;

minimize reward: - sum{j in A} mean[j]*alloc[j];

subject to risk_bound:
  sum{i in T} (sum{j in A} Rtilde[i,j]*alloc[j])^2 / card{T} <= s_max;

subject to tot_mass: sum{j in A} alloc[j] = 1;
```

Notons :

— Nous avons introduit la variable `alloc` pour indiquer le taux de ressources que l'on veut utiliser au niveau de chaque investissement.
— Le retour sur investissement est donné comme suit $\sum_{j \in A} mean_j * alloc_j$.
— Le risque encouru est donné par $\sum_{i \in T}(\sum_{j \in A} \tilde{R}_{ij} * alloc_j)^2 / card(T)$; où $\tilde{R}_{ij} = R_{ij} - mean_j$.
— On voit que la fonction objectif du problème est linéaire, alors que les contraintes sont quadratiques.

Pour enfin compléter le modèle, soit le jeu de données suivant :

```
set A := US_3-MONTH_T-BILLS US_GOVN_LONG_BONDS SP_500 WILSHIRE_5000;

param R:
US_3-MONTH_T-BILLS US_GOVN_LONG_BONDS SP_500 WILSHIRE_5000 :=
1984 1.103 1.159 1.061 1.030
1985 1.080 1.366 1.316 1.326
1986 1.063 1.309 1.186 1.161
1987 1.061 0.925 1.052 1.023
1988 1.071 1.086 1.165 1.179
1989 1.087 1.212 1.316 1.292
1990 1.080 1.054 0.968 0.938
1991 1.057 1.193 1.304 1.342
1992 1.036 1.079 1.076 1.090
1993 1.031 1.217 1.100 1.113
1994 1.045 0.889 1.012 0.999 ;
```

Comme nous l'avons vu dans le cours, ces problèmes sont difficiles à résoudre. La plupart des solveurs fournissent des solution locales, c'est-à-dire des solutions qui sont optimales localement à un voisinage. Justement, `Minos` fournit des solutions locales.

Il existe des solveurs qui trouvent des solutions globales, telles que `Baron`, `GlobSol`, etc. mais qui ne sont pas encore interfacés avec l'environnement `AMPL`.

3.8 Exercices compléments

3.8.1 Exercice

Etant donné deux variables x_1, x_2 binaires (i.e., $x_1 \in \{0,1\}$, $x_2 \in \{0,1\}$). Soit l'expression non linéaire de leur multiplication :

$$(P) \begin{cases} y = x_1 \times x_2 \\ x_1 \in \{0,1\}, x_2 \in \{0,1\}, y \in \{0,1\} \end{cases} \tag{3.9}$$

Nous proposons un système d'équations linéaires (Q)

$$(Q) \begin{cases} y \leq x_1 \\ y \leq x_2 \\ y \geq x_1 + x_2 - 1 \\ x_1 \in \{0,1\}, x_2 \in \{0,1\}, y \in \{0,1\} \end{cases} \tag{3.10}$$

Démontrer que (P) est équivalent à (Q) ?

3.8.2 Exercice

Soit le problème

$$(P) \begin{cases} \min \sum_{i=1..n} c_i x_i \\ \\ (\sum_{j=1..n} A_j x_j \leq b) \text{ OU } (\sum_{j=1..n} C_j x_j \leq d) \\ x_i \geq 0, i = 1..n \end{cases} \tag{3.11}$$

Le problème (P) n'est pas un PL car il contient la contrainte de disjonction "OU". Nous proposons un autre problème (Q)

$$(Q) \begin{cases} \min \sum_{i=1..n} c_i x_i \\ \sum_{j=1..n} A_j x_j \leq b + My \\ \sum_{j=1..n} C_j x_j \leq d + M(1-y) \\ x_i \geq 0, i = 1..n \\ y \in \{0,1\} \end{cases} \tag{3.12}$$

où M est un nombre très grand.

Démontrer que (P) est équivalent à (Q) ?

3.8.3 Exercice

Une firme *Penault* produit deux types de carrosseries C_1 et C_2. Nous nous intéressons à deux tâches indépendantes :

- *Peinture* : Cette firme ne pourra pas peindre C_2 à partir d'un maximum de 40 carrosseries C_1. Cette firme ne pourra pas aussi peindre C_1 à partir d'un maximum de 60 carrosseries C_2.
- *Fabrication* : Cette firme ne pourra pas produire C_2 à partir d'un maximum de 50 carrosseries C_1. Cette firme ne pourra pas aussi fabriquer C_1 à partir d'un maximum de 50 carrosseries C_2.

La carrosserie C_1 rapporte 300 DA et C_2 200 DA.

1. Modéliser sous forme d'un programme linéaire PL la recherche des quantités optimales x_1 de C_1 et x_2 de C_2 à produire pour maximiser le gain ? (N.B. Dans une tâche T donnée dans la firme, si la firme ne pourra pas produire C_2 à partir de k_1 C_1, et la firme ne pourra pas aussi produire C_1 à partir de k_2 C_2, la contrainte T peut être traduite en une seule contrainte linéaire $(x_1/k_1) + (x_2/k_2) \leq 1$.)
2. Tracer graphiquement le polyèdre décrivant le PL et extraire la solution optimale sans passer par l'algorithme du Simplexe ?
3. Soit la contrainte supplémentaire suivante : la firme *Penault* doit produire au moins 30 carrosseries C_1 et au moins 20 carrosseries C_2. Comment devient-il le PL ?

3.8.4 Exercice

Soit le programme en nombres entiers :

$$(P) \begin{cases} \min & \sum_{j=1..n} c_j x_j \\ \text{s.c.} & \sum_{j=1..n} A_{i,j} x_j \leq V_i, i = 1..m \\ & x_j \in \{0,1\}, j = 1..n \end{cases} \tag{3.13}$$

— Définir ce qu'est une relaxation du problème (P) ?
— Proposer une méthode utilisant la méthode du Simplexe qui permet la relaxation du problème (P) ?

3.8.5 Exercice

Soit le programme en nombres entiers relatif au problème du sac-à-dos :

$$(P) \begin{cases} \max & \sum_{j=1..n} c_j x_j \\ \text{s.c.} & \sum_{j=1..n} p_j x_j \leq V \\ & x_j \in \{0,1\}, j = 1..n \end{cases} \tag{3.14}$$

— Définir ce qu'est une relaxation du problème (P) ?
— Proposer une heuristique rapide de résolution approchée de (P) sans faire appel à la méthode du Simplexe ?

3.8.6 Exercice

Soit le programme en nombres entiers :

$$(P) \begin{cases} \min & \sum_{j=1..n} c_j x_j \\ \text{s.c.} & \sum_{j=1..n} A_{i,j} x_j \leq V_i, i = 1..m \\ & x_j \in \{0,1\}, j = 1..n \end{cases} \tag{3.15}$$

— Définir le principe d'évaluation (bounding) dans l'algorithme de branch& bound résolvant (P) ?

— Définir le principe de séparation (branching) dans l'algorithme de branch& bound résolvant (P) ?
— Comment sont combinés ces deux principes (i.e., séparation et évaluation) afin de résoudre (P) ?

3.8.7 Exercice

Soit le programme linéaire suivant :

$$\begin{array}{ll} \min & 2x_1 + 3x_2 \\ \text{s.c.} & \frac{1}{2}x_1 + x_2 \geq 1 \\ & \frac{2}{3}x_1 - x_2 \geq -2 \\ & x_1, x_2 \geq 0 \end{array} \tag{3.16}$$

1. Dessiner le polyèdre qui définit la forme géométrique des inéquations du problème (i.e. l'espace faisable) ?
2. Donner l'ensemble des sommets du polyèdre ?
3. Donner le sommet qui correspond à la solution optimale du problème ; tout en justifiant son optimalité ?

Chapitre 4

Introduction à l'optimisation continue

4.1 Préliminaires

4.1.1 Concepts de base d'optimisation continue

Définition 1 (problème d'optimisation continue (POC)). *Un problème général d'optimisation continue (POC) s'exprime comme suit : trouver des valeurs des variables de décision $x_1, ..., x_n$ telles que*

$$\begin{aligned} &minz = f(x_1, ..., x_n) \\ &g_i(x_1, ..., x_n) = 0, i = 1..m_e \\ &g_j(x_1, ..., x_n) \leq 0, j = m_e + 1..m. \end{aligned}$$

Définition 2 (région (ou espace) faisable). *L'ensemble des points $(x_1, ..., x_n)$ satisfaisant les m contraintes dans un POC est appelé espace faisable. Un point dans cet espace est dit point faisable, sinon on dit point infaisable.*

Définition 3 (minimum global et solution optimale). *Un point x^* de l'espace faisable pour lequel $f(x^*) \leq f(x)$ est vérifiés pour tous les points faisables x est dit minimum global (solution optimale) du POC.*

Définition 4 (minimum local et solution locale). *Un point faisable $x' = (x'_1, ..., x'_n)$ est un minimum local (solution locale) s'il existe un ϵ tel que $\forall x = (x_1, ..., x_n)$ avec*

$$\forall i, |x_i - x'_i| < \epsilon, f(x) \geq f(x').$$

Exemple : Si K unités de matières premières et L unités de travail sont utilisées, une entreprise peut produire KL unités d'un bien manufacturé. La matière première peut être acheté à 4 \$/unité et la main-d'oeuvre à 1 \$/unité. Un total de 8 \$ est disponible pour l'achat de la matière première et de main-d'oeuvre. Comment l'entreprise peut-elle maximiser la quantité de bien pouvant être fabriquée ?

$$\begin{aligned} &maxz = KL \\ &4K + L \leq 8 \\ &K, L \geq 0 \end{aligned}$$

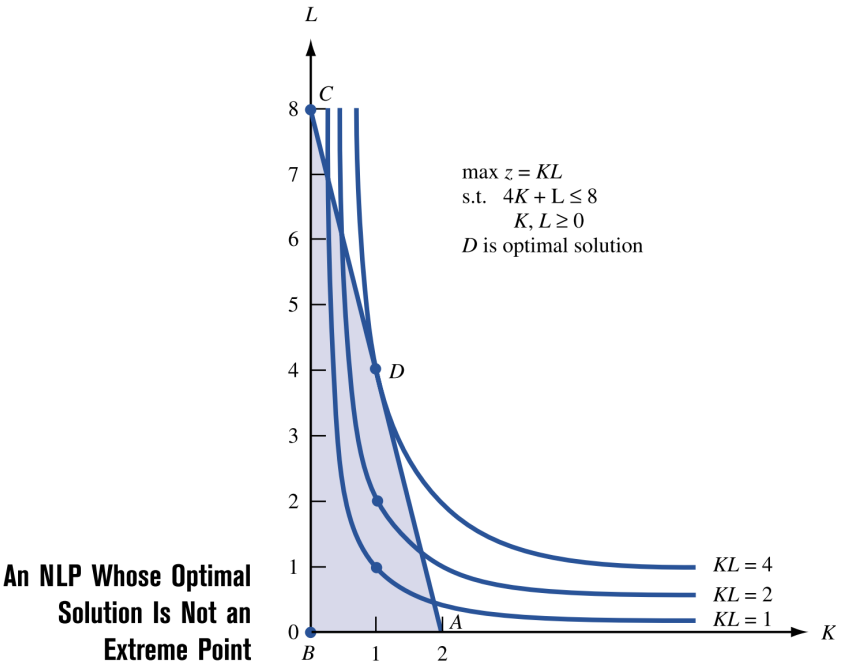


FIGURE 4.1 – Espace faisable et solution optimale.

4.1.2 Fonctions convexes

Définition 5 (ensemble convexe). *Un ensemble s est convexe si $\forall x', x'' \in s$, tout point du segment liant x' à x'' appartient à s ; c'est-à-dire*

$$\forall c \in [0,1], cx' + (1-c)x'' \in s.$$

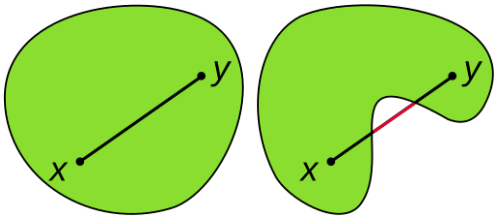


FIGURE 4.2 – Illustration d'un ensemble convexe vs non-convexe

Définition 6 (fonction convexe et fonction concave). *Soit $f(x_1, ..., x_n)$ une fonction définie sur un ensemble convexe s est convexe (resp. concave)si $\forall x' \in s, x'' \in s$*

$$f(cx' + (1-c)x'') \leq cf(x') + (1-c)f(x'')$$

$$(resp.\ f(cx' + (1-c)x'') \geq cf(x') + (1-c)f(x''))$$

pour tout $c \in [0,1]$.

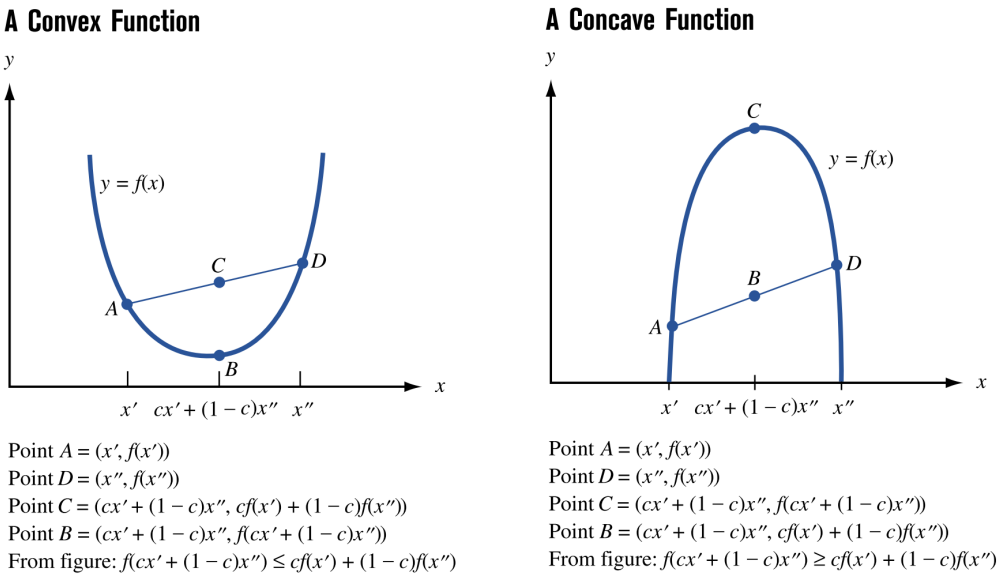


FIGURE 4.3 – Fonctions convexes et fonctions concaves.

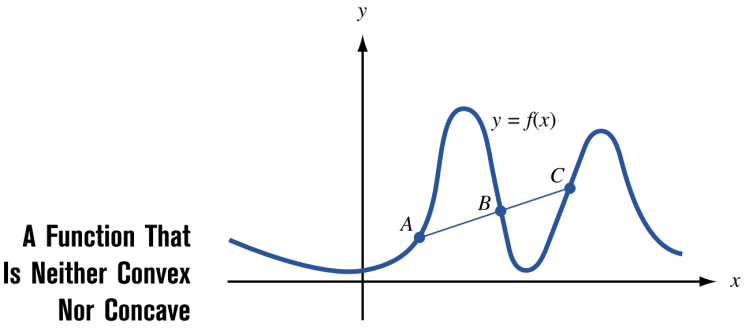


FIGURE 4.4 – Fonctions convexes et fonctions concaves.

Théorème 1 (optimum d'un POC convexe). *Supposons que l'espace faisable s d'un POC est convexe. Si $f(x)$ est convexe dans s, alors tout minimum local du POC est un minimum global.*

Preuve : exercice !

Théorème 2 (convexité et concavité). *Supposons que $f''(x)$ existe pour tout x dans un espace convexe, alors $f(x)$ est convexe (resp. concave) dans s ssi $\forall x, f''(x) \geq 0$ (resp. $f''(x) \leq 0$).*

Définition 7 (matrice hessienne). *La matrice hessienne de la fonction $f : x = (x_1, ..., x_n) \rightarrow f(x) = f(x_1, ..., x_n)$ est la matrice carrée $n \times n$ où la (i, j)ième entrée est définie par $\frac{\partial^2}{\partial x_i \partial x_j}$.*

4.1.3 Illustration : résoudre un POC sans contraintes avec une seule variable

Soit le POC suivant

$$min_{x\in[a,b]} f(x).$$

Naïvement, on peut calculer l'ensemble de tous les minimums locaux puis on prend le plus petit. Nous avons trois cas :

1. points x où $a < x < b$ et $f'(x) = 0$ (appelés points stationnaires)
2. points x où $f'(x)$ n'existe pas
3. les points extremums a et b de $[a, b]$

Parmi les points stationnaires, on prendrait seulement ceux qui sont des minimums locaux, car on peut avoir $f'(x) = 0$ sans que le point soit un minimum local : $f'(x) = 0$ est une condition nécessaire mais insuffisante pour que le point x soit un minimum local. Nous avons donc besoin de propriétés mathématiques pour localiser les minimums locaux. Et en plus, nous devons aussi considérés les valeurs de f sur les points extremums a et b et aux points où les dérivées ne sont pas disponibles car ces derniers peuvent contenir le minimum global.

Voici un théorème qui donne quelques propriétés pour localiser un minimum local.

Théorème 3 (localisation des minimums locaux). *Si $f'(x_0) = 0$ et $f''(x_0) > 0$ alors x_0 est un minimum local.*

Si $f'(x_0) = 0$ et $f''(x_0) < 0$ alors x_0 est un maximum local.

On veut résoudre le POC unidimensionnel sans contraintes

$$max_{a\leq x\leq b} f(x)$$

en se privant d'utiliser la dérivée $f'(x)$.

Dans cette section, on décrit comment résoudre ce POC si la fonction est unimodulaire.

Définition 8 (fonction unimodulaire). *La fonction $f : x \to f(x)$ est unimodulaire dans $[c, d]$ si $\exists x^* \in [a, b]$ où $f(x)$ est strictement croissante dans $[a, x^*)$ et décroissante dans $[x^*, b]$.*

Ce problème peut être résolu simplement par dichotomie. La procédure de résolution se présente comme suit :

1. Soient x_1 et x_2 des points distinct dans $[a, b]$.
2. Si $f(x_1) = f(x_2)$ alors nécessairement le maximum est dans l'intervalle $[a, x_2]$.
3. Si $f(x_1) < f(x_2)$ alors nécessairement le maximum est dans l'intervalle $[x_1, b]$.
4. Si $f(x_1) > f(x_2)$ alors nécessairement le maximum est dans l'intervalle $[a, x_2]$.
5. Relancer ce traitement sur l'intervalle restant jusqu'à ce que sa largeur soit inférieure à la précision exigée par l'utilisateur.

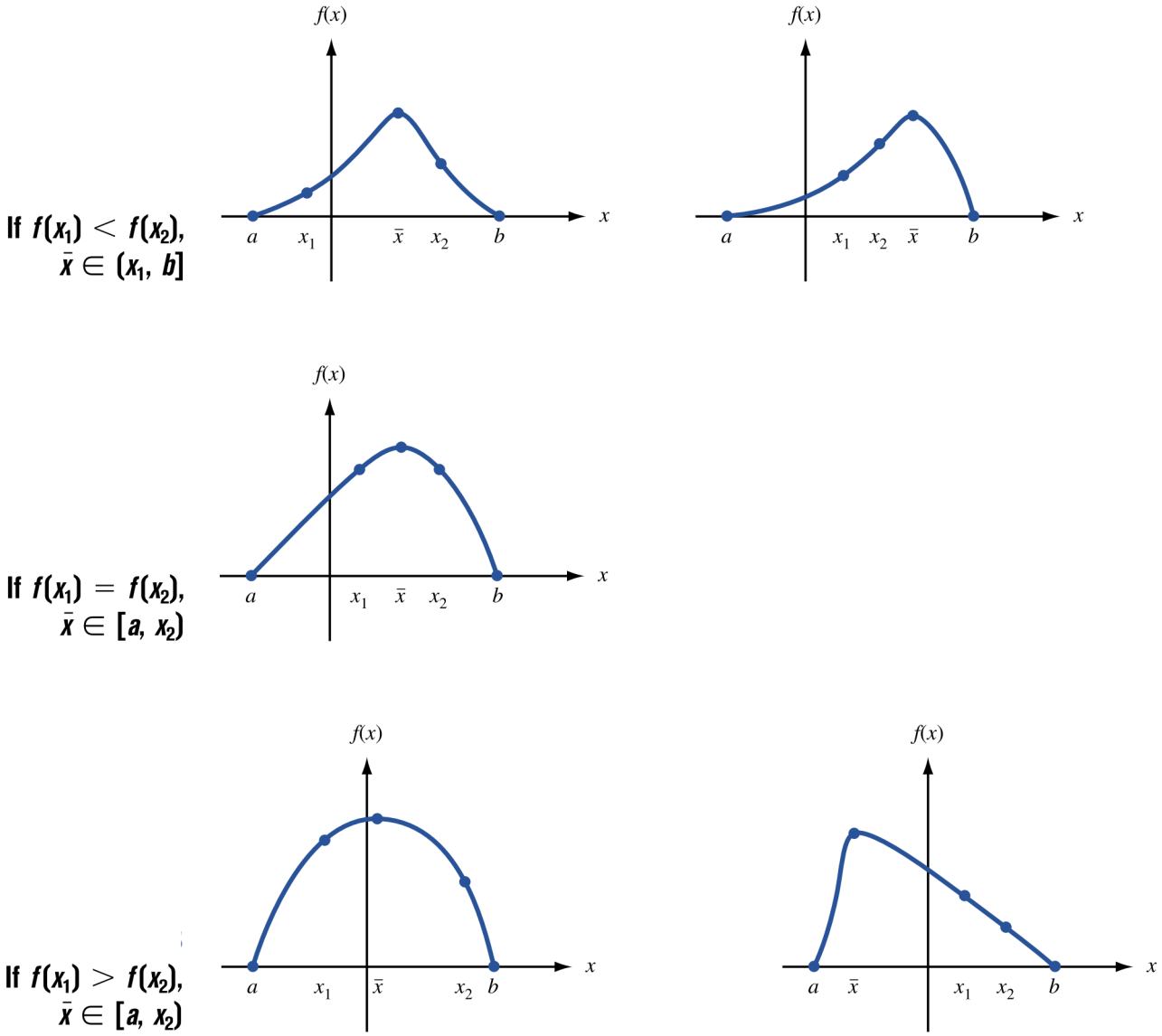


FIGURE 4.5 – Illustration des 3 cas d'optimisation/unimodulaire.

Cette recherche est dite méthode du nombre d'or si à chaque itération on prend $x_2 = a + (b - a)0.618$ et $x1 = b - (b - a)0.618$.

0.618 provient du nombre d'or (ou section dorée, proportion dorée), qui est une proportion définie initialement en géométrie comme l'unique rapport a/b entre deux longueurs a et b telles que le rapport de la somme $a + b$ des deux longueurs sur la plus grande (a) soit égal à celui de la plus grande (a) sur la plus petite (b), ce qui s'écrit :

$$\frac{a+b}{a} = \frac{a}{b}$$

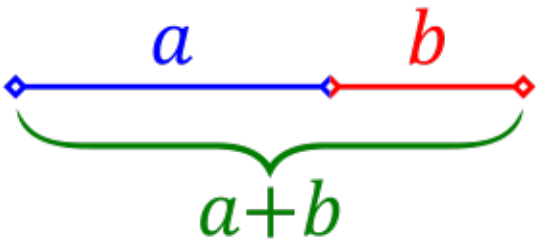


FIGURE 4.6 – Illustration du nombre d'or.

On a donc

$$\frac{a}{b} = r = 1 + \frac{1}{r}.$$

La seule solution de ce problème $r^2 = r + 1$ est $r = \frac{5^{1/2}+1}{2} = 1.6180339887$.
TP : mettre en oeuvre cet algo !

4.1.4 Algorithmes et convergence

Définition 9 (vitesse de convergence). *1. si* $lim_{k\to\infty}\frac{\|u^{k+1}-u^*\|}{\|u^k-u^*\|} = \alpha < 1$, *on dit que la convergence est linéaire et* α *est la taux de convergence associé.*

2. si $lim_{k\to\infty}\frac{\|u^{k+1}-u^*\|}{\|u^k-u^*\|} = 0$, *on dit que la convergence est superlinéaire.*

3. si $\exists\gamma, lim_{k\to\infty} sup\frac{\|u^{k+1}-u^*\|}{\|u^k-u^*\|^\gamma} = \alpha < 1$, *on dit que la convergence est d'ordre* γ. *En particulier si* $\gamma = 2$, *on parle de vitesse de convergence quadratique* ($\gamma = 3$, *cubique).*

On peut définir ([?, ?])

$$d_k = -\log_{10}|u^k - u^*|,$$

cette quantité, à une approximation près, représente le nombre de chiffres décimaux exacts de u_k.

Exemple 1. *[?]*
Soit la suite :

$$u_k = 12 + \frac{1}{n}$$

qui converge vers 12. *Nous avons :*

$$\begin{array}{rclrcl} u_{100} & = & 12.01 & d_{100} & = & 2 \\ u_{1000} & = & 12.001 & d_{1000} & = & 3 \\ u_{10000} & = & 12.0001 & d_{10000} & = & 4 \end{array}$$

Lorsque k est suffisamment grand, on peut trouver une constante C telle que :

$$|u_{n+1} - u^*| \approx C|u_k - u^*|^r$$

Nous avons donc $d_{n+1} = r \times d_n - \log_{10}(C)$. C'est-à-dire qu'à chaque itération, nous multiplions par environ r le nombre de chiffres exacts et nous ajoutons $R = -\log_{10}(C)$. Si $r = 1$, nous ne faisons qu'ajouter R chiffres décimaux par itération.

Exemple 2. *[?]*
Si $C = 0.999$ *alors* $R = 1/2500$. *Cela signifie que l'on obtient un nouveau chiffre exact toutes les 2500 itérations. Par contre, si* $r = 1.01$, *on multiplie par* 2 *le nombre de chiffres exacts toutes les* 70 *itérations ; car pour avoir* $2 = 1.01^n$, *on obtient* $n = log(2)/log(1.01) = 70$. *Si* $r = 2$, *on multiplie par 2 le nombre de chiffre significatifs à chaque itération* $n = 1$.

Ces exemples montrent bien l'intérêt d'utiliser les suites d'un ordre supérieur à 1. C'est pour cette raison que la convergence d'ordre deux ou *quadratique* est caractérisée de convergence *rapide*.

4.2 Optimisation sans contraintes

Dans cette section, nous nous intéressons au problème suivant

$$min_{x \in I\!R^n} f(x) \tag{4.1}$$

où $x \in I\!R^n$ est un vecteur réel avec $n \geq 1$ et $f : I\!R^n \to I\!R$ est une fonction continue et continûment dérivable.

Nous supposons que les premières et secondes dérivées de $f(x_1, x_2, ..., x_n)$ existent et sont continues sur tous les points. Soit $\frac{\partial f}{\partial \tilde{x}_i}$ la dérivée partielle de $f(x_1, x_2, ..., x_n)$ relativement à x_i, évaluée au point $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n})$.

Une condition nécessaire pour que $\tilde{x} = \tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}$ soit un minimum local de (4.1) est donnée par le théorème suivant :

Théorème 4. *Si* $\tilde{x}$ *est un minimum local de (4.1), alors* $\frac{\partial f(\tilde{x})}{\partial x_i} = 0$.

La preuve de ce théorème vient du fait que si $\tilde{x}$ est un minimum local alors il a un voisinnage où la fonction objectif prend son min en ce point. Supposons que le point en question n'annule pas la dérivée de la fonction objectif, alors nécessairement la fonction objectif est monotone en ce point faisant ainsi diminuer la fonction objectif ; ce qui contredit que le point est optimal dans son voisinnage.

Définition 10. *Un point* $\tilde{x}$ *ayant* $\frac{\partial f}{\partial x_i} = 0$ *pour tout* $i = 1, 2, ..., n$ *est dit un point stationnaire de* f.

Définition 11.
- *Le i-ème principal mineur (principal minor) d'une matrice* $n \times n$ *est le déterminant de la matrice* $i \times i$ *obtenue en supprimant* $n - i$ *lignes et les* $n - i$ *correspondantes colonnes de la matrice.*
- *Le k-ème principal mineur dominant (leading principal minor) d'une matrice* $n \times n$ *est le déterminant de la matrice* $k \times k$ *obtenue en supprimant les dernières* $n - k$ *lignes et colonnes de la matrice. On note* $H_k(x_1, ..., x_n)$ *le kième principal mineur dominant au point* $(x_1, ..., x_n)$.

Par exemple, soit $f(x_1, x_2) = x_1^2 + 2x_1x_2 + x_2^2$, alors $H_1(x_1, x_2) = 6x_1$ et $H_2(x_1, x_2) = 12x_1 - 4$.

Remarque 1 (compléments sur la convexité et la concavité)**.** *Supposons que* $f(x_1, x_2, ..., x_n)$ *a les dérivées secondes continues.*

— *$f(x_1, x_2, ..., x_n)$ est une fonction convexe dans S si et seulement si pour tout $x \in S$, tous les principaux mineurs de H sont non-négatifs.*
— *$f(x_1, x_2, ..., x_n)$ est une fonction concave dans S si et seulement si pour tout $x \in S$ et $k = 1, 2, ..., n$, tous les principaux mineurs non-nuls de H ont le même signe que -1^k.*

(exemples : page 655 du Winston)

Le théorème suivant montre quelques conditions suffisantes pour qu'un point soit un extrémum local à partir de la matrice hessienne.

Théorème 5. — *Si $H_k(\tilde{x}) > 0, k = 1..n$, alors le point $\tilde{x}$ est un minimum local de (4.1).*
— *Si $H_k(\tilde{x}), k = 1..n$, est non-nul et a le même signe que -1^k, alors le point $\tilde{x}$ est un maximum local de (4.1).*
— *Si $H_k(\tilde{x}) \neq 0, k = 1..n$, et les deux conditions précédentes ne sont pas vérifiées, alors le point $\tilde{x}$ n'est pas un extrémum local de (4.1).*

(exemples page 670 du Winston)

Si un point stationnaire n'est pas un minimum local, il est dit un point d'inflexion (saddle point).

Nous pouvons aussi exploiter les propriétés de convexité et de concavité de la fonction objectif pour détecter des minimums globaux (voir la section 4.1.2).

Remarque 2 (rappel sur l'optimum d'un POC convexe)**.** *Supposons que l'espace faisable s d'un POC est convexe. Si $f(x)$ est convexe dans s, alors tout minimum local du POC est un minimum global.*

Exemple 14 ([?])**.** *Nous avons n points mesurant une grandeur physique m fois dans le temps $t_1, ..., t_m$. A partir de considérations théoriques nous savons que la fonction de cette grandeur physique est*

$$\Phi(t, x) = x_1 + x_2 e^{(x_3 - t)^2 / x_4} + x_5 cos(x_6 t).$$

$x_1 ... x_6$ sont des paramètres. Soit $r_j(x) = y_j - \phi(t_j, x)$ mesurant l'écart entre la mesure expérimentale et la valeur théorique. Trouver les $x_1, ..., x_6$ tels que

$$min_x f(x) = r_1^2(x) + ... + r_m^2(x).$$

Nous allons tout d'abord commencer par voir les méthodes locales de descente, puis nous allons aborder les méthodes globales par intervalles.

Les méthodes les plus efficaces d'optimisation continue sans contraintes utilisent le gradient de la fonction objectif. La méthode la plus simple est celle de la plus grande descente et les plus raffinées et performantes sont celles de quasi-Newton et Newton (secant methods).

Toutes ces méthodes sont basées sur la stratégie de recherche linéaire que nous introduisons ci-dessous.

FIGURE 4.7 – Recherche linéaire

$\lambda_k := 1$;
while $(f(x_k + \lambda_k/2d_k) < f(x_k + \lambda d_k))$ **do**
$\lambda_k := \lambda_k/2$;
end while

FIGURE 4.8 – Schéma des algorithmes de stratégie linéaire

Initialisation de x_0 et d_0 ;
k := 0 ;
do
Déterminer λ_k tel que $minf(x_k + \lambda_k d_k)$
Calculer le nouveau point $x_{k+1} := x_k + \lambda_k d_k$
Calculer la nouvelle direction d_{k+1} ;
while $(|x_{k+1} - x_k| < \epsilon)$

4.2.1 Recherche linéaire

L'algorithme choisi une direction d_k et cherche au long de cette direction à partir de l'itéré courant x_k un nouveau itéréx_{k+1} minimisant la fonction objectif. La structure générale de cet algorithme est donnée dans la Figure 4.2.1.

Nous voulons résoudre $minf(x_k + \lambda_k d_k)$ avec $\lambda_k \in [0, 1]$.

Une solution approchée de ce problème peut être obtenue avec la procédure donnée dans la Figure 4.2.1.

Cette recherche est de convergence linéaire. Elle peut être raffinée en construisant un modèle quadratique de $f(x_k + \lambda_k d_k)$ en fonction de λ_k. Ce modèle quadratique est de la forme

$$P(\lambda) = f_1 + h_1\lambda + h_3\lambda(\lambda - \lambda_2)$$

qui interpole la fonction au 3 points $\alpha = 0, \alpha = \alpha_2, \alpha = \alpha_3$; où

$$\begin{aligned} f_i &= f(x_k + \lambda_i d_k) \\ h_1 &= (f_2 - f_1)\lambda_2 \\ h_2 &= (f_3 - f_2)/(\lambda_3 - \lambda_2) \\ \lambda_2 &= \lambda_3/2 \\ h_3 &= (h_2 - h_1)/\lambda_3 \end{aligned}$$

On sait que la solution de ce modèle est $\lambda_3 = 0.5(\lambda_2 - h_1/h_3)$. Cette dernière solution peut donner une meilleure solution que que le choix naïf $\lambda/2$. Quand cette méthode est utilisée dans un quasi-Newton, on doit prendre en considération dans le choix de λ le fait que H doit rester définie positive.

4.2.2 Méthode de la plus grande descente

Ce sont les méthodes les plus simples. La recherche de la direction est faite suivant la décroissance de f qui est dans l'opposé de $\Delta f(x)$

$$x_{k+1} := x_k - \lambda_k \Delta f(x_k).$$

où $\Delta f(x) = [\frac{\partial f(x)}{\partial x_1}, ..., \frac{\partial f(x)}{\partial x_n}]$ est le vecteur gradient de $f(x)$.

On peut se poser la question : quel est la grandeur d'influence de λ_k ?

$$\frac{\partial f(x_{k+1})}{\partial \lambda_k} = \frac{\partial f(x_k - \lambda_k \Delta(f(x_k)))}{\partial \lambda_k} = 0$$
$$\Rightarrow -\Delta f(x_{k+1}) \partial f(x_k) = 0$$

λ_k est donc choisie telle que $\Delta f(x_{k+1})$ et $\Delta f(x_k)$ soient orthogonales.

Cette méthode est de convergence linéaire [**?**] ; mais elle a de bonne propriétés de convergence globale.

4.2.3 Méthode de quasi-Newton

La méthode la plus efficace et celle des méthodes de quasi-Newton. Ces méthodes construisent une approximation quadratique :

$$min_x \frac{1}{2} x^t H x + C^T x + b$$

où la matrice H est semi-définie positive.

Rappel 1 (matrice semi-définie positive (SDP))**.** *Une matrice $A \in \mathbb{R}^{n \times n}$ est dite semi-définie postive si*

$$\forall x \in \mathbb{R}^n \Rightarrow x^T A x \geq 0.$$

Elle est dite définie positive si

$$\forall x \in \mathbb{R}^n \Rightarrow x^T A x > 0.$$

Une condition nécessaire de l'optimalité d'une solution x^* de cette approximation quadratique est que les dérivées partielles soient toutes égales à zéro

$$\Delta f(x^*) = H x^* + C = 0.$$

Les méthodes de type Newton calculent H directement, ce qui est très couteux si le nombre de variables est important.

Plusieurs méthodes ont ainsi été développées pour approximer directement l'inverse de H avec seulement les valeurs du gradient. Deux formules ont été proposées, celle de DFP (Davidon, Fletcher et Powell) et BFGS (Broyden, Fletcher, Goldfard et Shanno).

La formule de BFGS est

$$H_{k+1} := H_k + \frac{q_k q_k^T}{q_k s_k} + \frac{H_k^T S_k^T S_k H_k}{S_k^T H_k s_k}$$

où $s_k := x_{k+1} - x_k$ et $q_k := \Delta f(x_{k+1}) - \Delta f(x_k)$.

DFP est similaire à BFGS en interchangant q_k et s_k. H_0 est fixée à n'importe quelle matrice symétrique semi-définie positive, par exemple la matrice identité. BFGS a les mêmes propriétés théoriques que DFP. Mais en pratique $BFGS$ est préférée à DFP car elle est plus stable numériquement (moins sensible aux erreurs d'arrondi).

L'algorithme général de quasi-Newton est celui de la stratégie linéaire où :

$$d_{k+1} := \ - H_{k+1}\Delta f(x_k)$$

Il a été prouvé que quasi-Newton est de convergence superlinéaire ; mais elle est moins bonne au niveau convergence globale que les méthodes de descente du gradient.

4.2.4 Moindres carrés

La stratégie de recherche linéaire de quasi-Newton peut être utilisée pour résoudre le problème d'optimisation particulier des moindres carrés

$$LS \quad minf(x) = \frac{1}{2}\|F(x)\| = \frac{1}{2}\sum F_i(x)^2.$$

Ce problème apparaît souvent dans plein d'applications pratiques particulières dans l'interpolation à partir de données expérimentales. Ce problèmes peut être résolu simplement avec un quasi-Newton. Les propriétés mathématiques de ce problème permettent une meilleure résolution.

La méthode qui tire profit de ces propriétés et qui est la plus utilisée sur les moindres carrés est celle de Levenberg-Marquardt (LM).

LM prend comme direction de recherche la solution d_k de l'équation

$$[J(x_k)^T J(x_k) + \lambda_k I]d_k = -J(x_k)F(x_k).$$

La solution recherche de (LS) est atteinte quand $F_i(x) = O, i = 1..m$.

La formule de Taylor permet de le ramener à :

$$F_i(x_k) + J(x_k)x = 0$$
$$\Rightarrow J_i(x_k).x = -F_i(x_k)$$

Puisque le système n'est pas carré, en le rend carré en le multipliant par la transposée

$$\Rightarrow J_i(x_k)^T J_i(x_k)x = -J_i(x_k)^T F_i(x_k).$$

Résoudre ce dernier système permet d'avoir la direction de quasi-Newton car $J_i(x_k)^T J_i(x_k)$ est le hessien de $\sum F_i(x)^2$.

En d'autres termes, plus λ_k augmente, LM prend comme direction de recherche la solution d_k de l'équation

$$[J(x_k)^T J(x_k) + \lambda_k I]d_k = -J(x_k)F(x_k).$$

plus on se rapproche de la méthode de descente et ainsi obtenir une meilleure convergence globale.

Donc quand on n'a pas la convergence de quasi-Newton, on augmente λ_k pour converger avec la méthode de la plus grande descente.

4.3 Optimisation avec des contraintes et une fonction objectif : le cas général

Un problème général d'optimisation continue (POC) s'exprime comme suit : trouver des valeurs des variables de décision $x_1, ..., x_n$ telles que :

$$\begin{array}{l} minz = f(x_1, ..., x_n) \\ g_i(x_1, ..., x_n) = b_i, i = 1..m_e \\ g_j(x_1, ..., x_n) \leq b_j, j = m_e + 1..m. \end{array} \tag{4.2}$$

4.3.1 Les multiplicateurs de Lagrange

Les multiplicateurs de Lagrange peuvent être utilisés quand les contraintes sont des égalités. Nous considérons donc le problème d'optimisation (POC) suivant :

$$\begin{array}{l} minz = f(x_1, ..., x_n) \\ c_i : g_i(x_1, ..., x_n) = b_i, i = 1..m_e \end{array} \tag{4.3}$$

Pour résoudre (4.3), on associe à chaque contrainte c_i un multiplicateur λ_i, et on forme ainsi le Lagrangien :

$$L(x_1, x_2, ..., x_n, \lambda_1, \lambda_2, ..., \lambda_m) = f(x_1, x_2, ..., x_n) + \sum_{i=1}^{m} \lambda_i [b_i - g_i(x_1, x_2, ..., x_n)] \tag{4.4}$$

Nous cherchons à trouver un point $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}, \tilde{\lambda_1}, , \tilde{\lambda_2}, ..., \tilde{\lambda_m})$ qui minimise $L(x_1, x_2, ..., x_n, \lambda_1, \lambda_2, ..., \lambda_m)$. Souvent $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}, \tilde{\lambda_1}, , \tilde{\lambda_2}, ..., \tilde{\lambda_m})$ résout aussi (4.3). Si $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}, \tilde{\lambda_1}, , \tilde{\lambda_2}, ..., \tilde{\lambda_m})$ minimise L, alors

$$\frac{\partial L}{\partial \lambda_i} = b_i - g_i(x_1, x_2, ..., x_n) = 0$$

Ici $\frac{\partial L}{\partial \lambda_i}$ est la dérivée partielle de L par rapport à λ_i. Ceci montre bien que le point en question satisfait d'une façon optimale(4.3). Pour montrer que $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n})$ résout (4.3), soit un point quelconque de l'espace faisable de (4.3). Puisque $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}, \tilde{\lambda_1}, \tilde{\lambda_2}, ..., \tilde{\lambda_m})$ minimise L, alors pour tout $\lambda'_i, i = 1..m$, nous avons :

$$L(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}, \tilde{\lambda_1}, , \tilde{\lambda_2}, ..., \tilde{\lambda_m}) \leq L(x'_1, x'_2, ..., x'_n, \lambda'_1, , \lambda'_2, ..., \lambda'_m)$$

Puisque $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n})$ et $(x'_1, x'_2, ..., x'_n)$ sont tous les deux faisables, alors tous les facteurs des multiplicateurs sont nuls. On obtiens ainsi $f(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}) \leq f(x'_1, x'_2, ..., x'_n)$.
Ceci montre que $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n})$ est bien la solution de (4.3). En bref, si $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}, \tilde{\lambda_1}, \tilde{\lambda_2}, ..., \tilde{\lambda_m})$ résout le problème d'optimisation sans contraintes

$$minL(x_1, x_2, ..., x_n, \lambda_1, \lambda_2, ..., \lambda_m) \tag{4.5}$$

alors $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n})$ résout aussi (4.3).

A partir du chapitre **??**, nous savons qu'une condition nécessaire pour que $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}, \tilde{\lambda_1}, \tilde{\lambda_2}, ..$ résolve (4.5) est qu'au point $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n}, \tilde{\lambda_1}, \tilde{\lambda_2}, ..., \tilde{\lambda_m})$ on doit avoir

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial x_2} = ... = \frac{\partial L}{\partial x_n} = \frac{\partial L}{\partial \lambda_1} = \frac{\partial L}{\partial \lambda_2} = ... = \frac{\partial L}{\partial \lambda_n} = 0 \tag{4.6}$$

Le théorème suivant donne des conditions suffisantes pour qu'un tel point soit un minimum global.

Théorème 6. *Si $f(x_1, x_2, ..., x_n)$ est une fonction convexe et chaque $g_i(x_1, ..., x_n)$ est une fonction linéaire, alors tout point $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n})$ satisfaisant (4.6) est une solution optimale.*

Les variables λ_i ont une interprétation intéressante comme étant les coûts marginaux (shadow prices) associés à chacune des des contraintes. Si la partie gauche d'une contrainte $g_i(x_1, x_2, ..., x_n) = b$ croit de δ_i, alors la valeur de la fonction objectif croit de $\delta_i \lambda_i$.

4.3.2 Les conditions de Kuhn-Tucker

Nous discutons dans cette section des conditions nécessaires et suffisantes pour que $(\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n})$ soit une solution optimale d'un POC avec des contraintes d'inégalité :

$$\begin{array}{l} minz = f(x_1, ..., x_n) \\ c_i : g_i(x_1, ..., x_n) \leq b_i, i = 1..m \end{array} \tag{4.7}$$

Pour que les résultats de cette section soient applicable, il qu'on ait seulement des contraintes d'inégalité inférieure. Les contraintes d'égalité ou d'inégalité supérieure peuvent être aisément ramenées en contraintes d'inégalité inférieure.

Le théorème suivant donne des conditions nécessaires pour qu'un point $\tilde{x} = (\tilde{x_1}, \tilde{x_2}, ..., \tilde{x_n})$ soit optimal pour le POC (4.7).

Pour que ce théorème soit applicable il faut que les fonctions g_i satisfassent des conditions de régularité (voir [?]). Quand les contraintes sont linéaires, ces conditions sont toujours satisfaites. Nous supposons dans la suite que les contraintes satisfassent toujours les conditions de régularité.

Théorème 7.

Comme au niveau des multiplicateurs de Lagrange de la section précédente, les multiplicateurs λ_i associés au conditions KT correspondent aux coûts marginaux des contraintes.

4.3.3 SQP : Sequential Quadratique Programming

4.3.4 TP

4.4 Résolution des équations non-linéaires

4.4.1 Algorithme de Newton

Les méthodes locales s'intéressent au problème suivant :

Définition 12 (local zero (LZ)). *Soit $f : \mathbb{R}^n \to \mathbb{R}^n$, trouver x^* une approximation locale du zéro de la fonction f ($y \in \mathbb{R}^n$ est un zéro de f si $f(y) = 0$).*

La méthode de Newton pour résoudre des systèmes d'équations consiste à appliquer le procédé itératif suivant en partant d'un point initial x^0 :

$$x^k = x^{k-1} - J(x^{k-1})^{-1} f(x^{k-1})$$

où $J(x)$ est la matrice jacobienne $n \times n$: $[\frac{\partial f_i(x)}{\partial x_j}, i = 1..m, j = 1..n]$.

Cette méthode est de convergence locale quadratique ; mais elle n'a pas de bonnes propriétés de convergence globale. C'est pour cette raison que l'on transforme le système d'équations en un problème d'optimisation pour bénéficier de ses méthodes de convergence globale. C'est l'objet de la sous-section suivante.

4.4.2 Transformation en un problèmes d'optimisation

Le système d'équations non-linéaires $f(x) = 0$ est transformé en un problème d'optimisation aux moindres carrés

$$min_x \sum_{i=1..m} f_i(x_1, ..., x_n)^2$$

Ainsi, on pourrait appliquer aisément les méthodes des moindres carrés du chapitre précédent. En raison de la particularité de ce problème, une méthode dédiée a été développée, l'algorithme de Levenberg-Marquardt-Method (LMM), pour exploiter ses propriétés.

LMM utilise la direction de recherche d_k qui est la solution du système linéaire :

$$(J(x_k)^T J(x) + \lambda_k I)d_k = -J(x_k)F(x_k)$$

où $J(x)$ est la matrice jacobienne$m \times n$: $[\frac{\partial f_i(x)}{\partial x_j}, i = 1..m, j = 1..n]$. Le scalaire λ_k contrôle la magnitude et la direction de d_k.

La valeur de λ_k est choisie pour garantir la convergence de l'ensemble du procédé. Quand λ_k est égale à zéro, l'algorithme se comporte comme un quasi-newton et bénéficie ainsi de sa bonne convergence locale. Plus λ_k est grand, plus l'algorithme se comporte comme l'algorithme de la plus grande descente et bénéficie ainsi de sa convergence globale. λ_k est initialisée à zéro, et si l'algorithme ne converge pas, λ_k est augmenté progressivement pour pouvoir ainsi converger globalement ; par la suite λ_k est diminuer pour converger rapidement une fois que les itérations précédentes ont permis de se rapprocher suffisamment de la solution. Une fois λ_k fixée, le vecteur d_k est obtenu en résolvant le système linéaire ci-dessus. Finalement, une stratégie de recherche linéaire (line-search strategy) est utilisée pour fixer α_k pour que $x_{k+1} = x_k + \alpha_k d_k$ fasse décroître la fonction objectif (la fonction aux moindres carrés).

Chapitre 5

Méthodes approchées et métaheuristiques

Chapitre 6

Méthodes d'optimisation pour l'IA et le deep-learning

6.1 Introduction aux architectures neuronales

6.2 Problématique d'optimisation dans l'apprentissage automatique

Chapitre 7

Annexes

7.1 Branch and Price et génération de colonnes

Soit le problème de plus courts chemins avec contraintes [?]. Soit le graphe de la Figure 7.1. En plus du cout c_{ij} attribué à chaque arc, nous disposons aussi d'un temps maximal t_{ij}. L'objectif est de trouver le plus court chemin du noeud 1 au noeud 6 sans excéder 14 unités de temps.

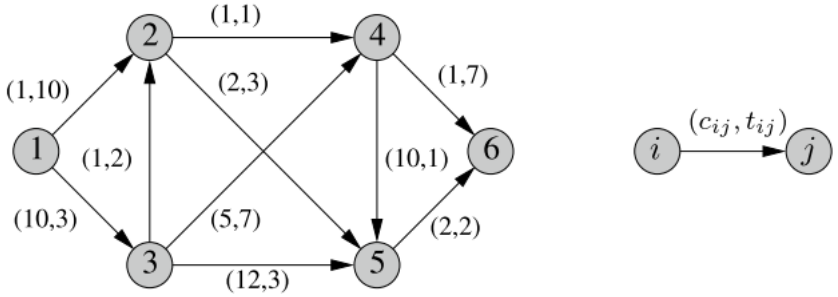


FIGURE 7.1 – Problème de plus courts chemins avec une contrainte de capacité [?]

On peut formaliser ce problème dans le problème IP (7.1) sur le domaine 0-1.

$$\begin{array}{ll} \text{minmisze} & z = \sum_{(i,j)\in A} c_{ij}x_{ij} \\ \text{subject to} & \sum_{j:(1,j)\in A} x_{1j} = 1 \\ & \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = 0, i = 2,3,4,5 \\ & \sum_{i:(i,6)\in A} x_{i6} = 1 \\ & \sum_{(i,j)\in A} t_{ij}x_{ij} \leq 14 \\ & x_{ij} \in \{0,1\}, (i,j) \in A \end{array} \tag{7.1}$$

En décortiquant le problème, on peut montrer qu'il existe neuf chemins possibles. Le chemin optimal est 13246. Comment obtenir une telle solution ? Au fait, le problème du plus court chemin est un problème facile, mais en l'augmentant avec la contrainte de capacité $\sum_{(i,j)\in A} t_{ij}x_{ij} \leq 14$, le problème devient NP-difficile.

Ignorons la contrainte de capacité $\sum_{(i,j)\in A} t_{ij}x_{ij} \leq 14$. Le problème devient un simple problème de plus courts chemins. Soit

$$X = \{x_{ij} \in 0..1 | (7.1) \text{sauf la contrainte de capacité} \leq 14\}.$$

Il est connu dans la théorie des flots que les solutions (sommets du polyèdre) définies par X correspondent aux chemins P du graphe allant de 1 à 6. Nous pouvons ainsi reformuler les contraintes du problème, mise à part la contrainte de capacité, comme suit :

$$\begin{array}{l} x_{ij} = \sum_{p\in P} x_{pij}\lambda_p, (i,j) \in A \\ \sum_{p\in P} \lambda_p = 1 \\ \lambda_p \geq 0, p \in P \end{array} \tag{7.2}$$

où x_{pij} correspond à une constante booléenne égale à 1 si l'arc (ij) appartient au chemin p, sinon 0.

En remplaçant (7.2) dans (7.1), nous obtenons :

$$\begin{array}{ll} \text{minmisze} & z = \sum_{p\in P}(\sum_{(i,j)\in A} c_{ij}x_{pij})\lambda_p \\ \text{subject to} & \sum_{p\in P}(\sum_{(i,j)\in A} t_{ij}x_{pij})\lambda_p \leq 14 \\ & \sum_{p\in P}\lambda_p = 1 \\ & \lambda_p \geq 0, p \in P \\ & x_{ij} = \sum_{p\in P} x_{pij}\lambda_p, (i,j) \in A \\ & x_{ij} \in \{0,1\}, (i,j) \in A \end{array} \tag{7.3}$$

Le problème (7.3) est dit le maître (master problème). Il y a autant de variables λ_p que de chemins P dans le graphe.

La contrainte

$$x_{ij} = \sum_{p\in P} x_{pij}\lambda_p, (i,j) \in A$$

est ignorée, car c'est une contrainte indépendante du problème, qui est satisfaite une fois le meilleure chemin est trouvé. Nous obtenons un problème avec 2 contraintes et 9 variables. Nous avons donc :

$$\begin{array}{ll} \text{minmisze} & z = \sum_{p\in P}(\sum_{(i,j)\in A} c_{ij}x_{pij})\lambda_p \\ \text{subject to} & \sum_{p\in P}(\sum_{(i,j)\in A} t_{ij}x_{pij})\lambda_p \leq 14 \\ & \sum_{p\in P}\lambda_p = 1 \\ & \lambda_p \geq 0, p \in P \end{array} \tag{7.4}$$

Dans le dual du problème (7.4), nous avons les deux variables π_1 et π_0 associées aux deux contraintes du problème. Ainsi, sur de très grands graphes, le nombre de variables devient prohibitif. D'où la nécessité à faire appel à la mécanique de la génération de colonnes. En conséquence, nous travaillerons que sur un ensemble restreint de colonnes, formant le problème maître restreint (restricted master problem) (MRP). Les variables hors (MRP) seront intégrées comme dans l'algorithme du simplex. Tant qu'il existe une variable qui n'est pas dans le (MRP) et qui pourrait l'améliorer, nous l'intégrerons. Ce processus itératif termine une fois qu'on ne trouve plus une colonne améliorante (c'est exactement le mécanisme de la colonne entrante et la colonne sortante du simplex).

On doit donc repérer la varibale entrante x_p ayant un cout réduit négatif :

$$\overline{c_p} = \sum_{(i,j)\in A} c_{ij}x_{pij} - \pi_1(\sum_{(i,j)\in A} t_{ij}x_{pij}) - \pi_0 < 0. \tag{7.5}$$

L'origine de cette formulation vient du tableau du simplex (**??**). En revoyant le déroulement de l'algorithme du simplex (voir la section **??**), nous avons :

$$Z = c_B B^{-1} b + (c_N - c_B B^{-1} A_N) X_N$$

Etant donnée la variable hors-base λ_p, son cout réduit est :

$$(c_N - c_B B^{-1} A_N)$$

où encore en utilisant la solution duale :

$$(c_N - \pi A_N)$$

Par analogie, on a le résultat final :

$$C_p = \sum_{(i,j)\in A} c_{ij} x_{pij}$$
$$A_0 = 1$$
$$A_1 = \pi_1(\sum_{(i,j)\in A} t_{ij}$$

La recherche de cette colonne entrante ayant le plus petit cout réduit revient à optimiser ce qu'on appelle le sous-problème (the subproblem). Dans notre cas, ce sera trouver le plus court chemin dans le graphe, avec un autre cout :

$$\overline{c_p}^* = min_{CC} \sum_{(i,j)\in A} (c_{ij} - \pi_1 t_{ij}) x_{ij} - \pi_0. \tag{7.6}$$

où CC correspond aux quatre contraintes :

$$\sum_{j:(1,j)\in A} x_{1j} = 1$$
$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = 0, i = 2,3,4,5$$
$$\sum_{i:(i,6)\in A} x_{i6} = 1$$
$$x_{ij} \in \{0,1\}, (i,j) \in A$$

Si $\overline{c_p}^* \geq 0$, alors il n'existe aucune colonne améliorante, d'où l'atteinte de l'optimalité. Sinon, la colonne trouvée est injectée dans le (RMP), et on réitère le processus. L'ensemble du processus itère 5 fois, comme illustré dans le Figure 7.2.

Pour trouver une solution entière, ce processus est intégré dans une procédure de type Branch and Bound, décrite au début de cette note.

Iteration	Master Solution	$\bar{z}$	π_0	π_1	$\bar{c}^\star$	p	c_p	t_p
BB0.1	$y_0 = 1$	100.0	100.00	0.00	−97.0	1246	3	18
BB0.2	$y_0 = 0.22, \lambda_{1246} = 0.78$	24.6	100.00	−5.39	−32.9	1356	24	8
BB0.3	$\lambda_{1246} = 0.6, \lambda_{1356} = 0.4$	11.4	40.80	−2.10	−4.8	13256	15	10
BB0.4	$\lambda_{1246} = \lambda_{13256} = 0.5$	9.0	30.00	−1.50	−2.5	1256	5	15
BB0.5	$\lambda_{13256} = 0.2, \lambda_{1256} = 0.8$	7.0	35.00	−2.00	0			
Arc flows: $x_{12} = 0.8, x_{13} = x_{32} = 0.2, x_{25} = x_{56} = 1$								

FIGURE 7.2 – Itérations de la génération de colonne sur le premier noeud du BB [?]

7.2 Branch and cut

Vu que l'énumération des variables entières peut être coûteuse, l'idée est d'ajouter des contraintes redondantes pour le modèle en nombres entiers, mais non pour la relaxation PL.

Exemple 15. *Considérons le programme mathématique*

$$\max_x \ 4x_1 + \frac{5}{2}x_2$$
$$x_1 + x_2 \leq 6$$
$$9x_1 + 5x_2 \leq 45$$
$$x_1,\ x_2 \in \mathcal{N}.$$

Le dictionnaire optimal correspondant à la relaxation linéaire de ce programme contient les deux contraintes

$$x_1 = \frac{15}{4} + \frac{5}{4}x_3 - \frac{1}{4}x_4,$$
$$x_2 = \frac{9}{4} - \frac{9}{4}x_3 + \frac{1}{4}x_4,$$

où x_3 et x_4 sont des variables d'écart. Puisque la variable de base x_1 n'est pas entière, cette solution de base n'est pas admissible. nous pouvons réécrire la première contrainte sous la forme

$$x_1 - \frac{5}{4}x_3 + \frac{1}{4}x_4 = \frac{15}{4}.$$

En utilisant l'identité

$$a = \lfloor a \rfloor + (a - \lfloor a \rfloor),$$

où $(a - \lfloor a \rfloor)$ représente la partie fractionnaire de a $(0 \leq a - \lfloor a \rfloor < 1)$, nous obtenons

$$x_1 + \left(\left\lfloor -\frac{5}{4} \right\rfloor + \frac{3}{4} \right) x3 + \left(\left\lfloor \frac{1}{4} \right\rfloor + \frac{1}{4} \right) x_4 = \left(\left\lfloor \frac{15}{4} \right\rfloor + \frac{3}{4} \right),$$

c'est-à-dire, en mettant tous les coefficients entiers à gauche et les coefficients fractionnaires à droite :

$$x_1 - 2x_3 - 3 = \frac{3}{4} - \frac{3}{4}x_3 - \frac{1}{4}x_4.$$

Puisque les variables x_3 et x_4 sont non négatives, la partie fractionnaire (constante du membre de droite) est inférieure à 1, le membre de droite est strictement inférieur à 1. Puisque le membre de gauche est entier, le membre de droite doit aussi être entier. Or un entier inférieur à 1 doit être inférieur ou égal à zéro. Nous en déduisons une contrainte additionnelle qui doit être satisfaite par toute solution admissible du problème originel, et que ne satisfait pas la solution de base courante :

$$\frac{3}{4} - \frac{3}{4}x_3 - \frac{1}{4}x_4 \leq 0.$$

En utilisant les identités $x_3 = 6 - x_1 - x_2$ et $x_4 = 45 - 9x_1 - 5x_2$, nous obtenons la coupe sous sa forme géométrique :

$$3x_1 + 2x_2 \leq 15.$$

Cette contrainte linéaire rend inadmissible la solution courante admissible, sans éliminer aucune autre solution entière. Si la solution du nouveau problème est entière, il s'agit de la solution optimale de notre problème. Sinon, nous construisons une nouvelle coupe et recommençons.

Exemple 16. *Reprenons le problème, illustré sur la Figure 7.3,*

$$\begin{aligned} \max\ & z = 3x_1 + 2x_2 \\ s.c.\ & 2x_1 + 3x_2 \leq 4, \\ & x_1, x_2 \text{ binaire.} \end{aligned}$$

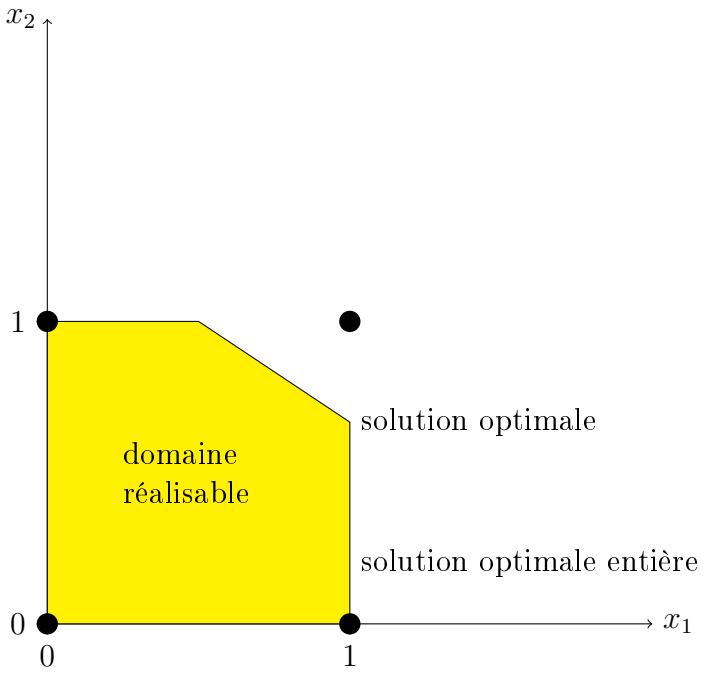


FIGURE 7.3 – Méthode de coupes

Les solutions réalisables sont $(0,0)$, $(1,0)$ *et* $(0,1)$.

Une contrainte redondante est :

$$x_1 + x_2 \leq 1.$$

Suite à l'ajout de la contrainte redondante, comme représenté sur la Figure 7.4, le problème est résolu à la racine.

Il y a plusieurs algorithmes permettant de générer de telles inégalités redondantes, appelées coupes. Mais il est rare que leur ajout permette de résoudre le probleme à la racine. L'ajout de coupes permet toutefois de réduire le nombre de sous-problèmes traités par l'algorithme de Branch-and-Bound. Nous pouvons même ajouter des coupes pour chaque sous-problème (pas seulement à la racine) : nous obtenons alors un algorithme de branch-and-cut.

Soit le problème [1] :

$$\begin{array}{ll} \text{minimize} & z = -6x_1 - 5x_2 \\ \text{subject to} & 3x_1 + x_2 \leq 11 \\ & -x_1 + 2x_2 \leq 5 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \text{variables entières.} \end{array} \tag{7.7}$$

Ce problème est illustré dans la Figure 7.6

1. www.acsu.buffalo.edu/ nagi/courses/684/branch.pdf ?

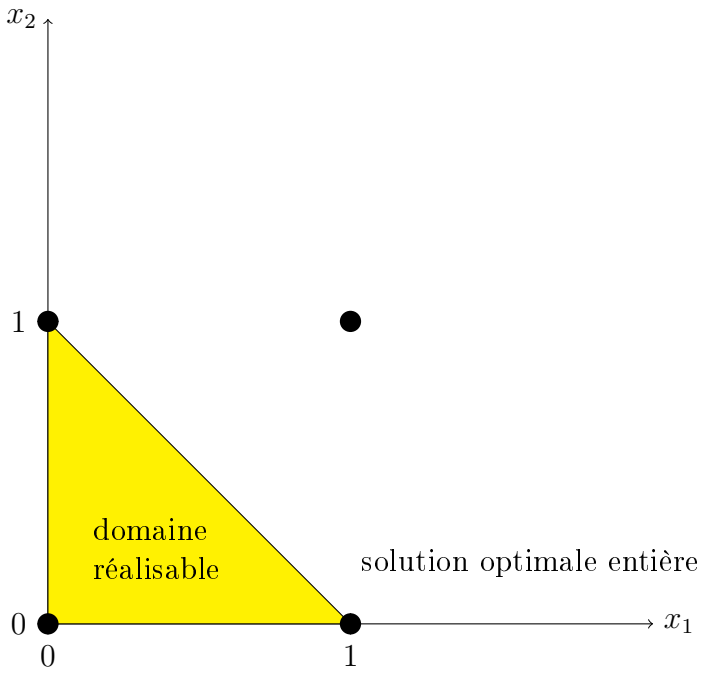


FIGURE 7.4 – Méthode de coupes : ajout d'une contrainte

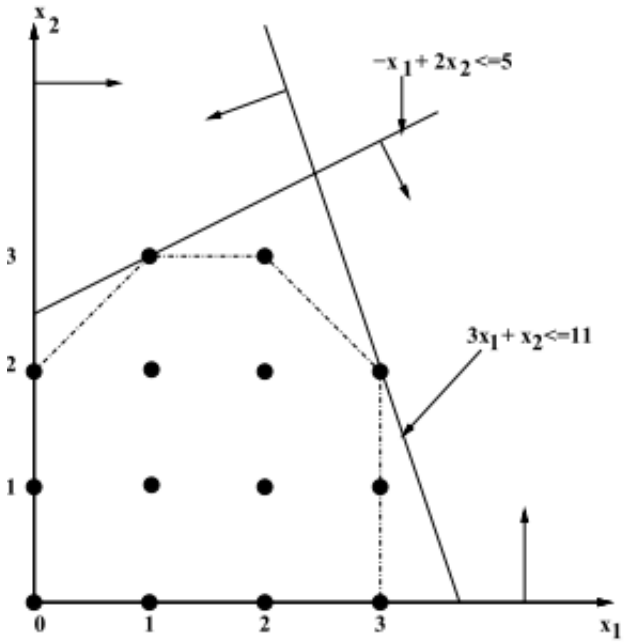


FIGURE 7.5 – Illustration du problème (7.7)

La procédure Branch and Bound sur le problème (7.7) est illustrée dans la Figure 7.6 avec la génération de la coupe $2x_1 + x_2 \leq 7$.

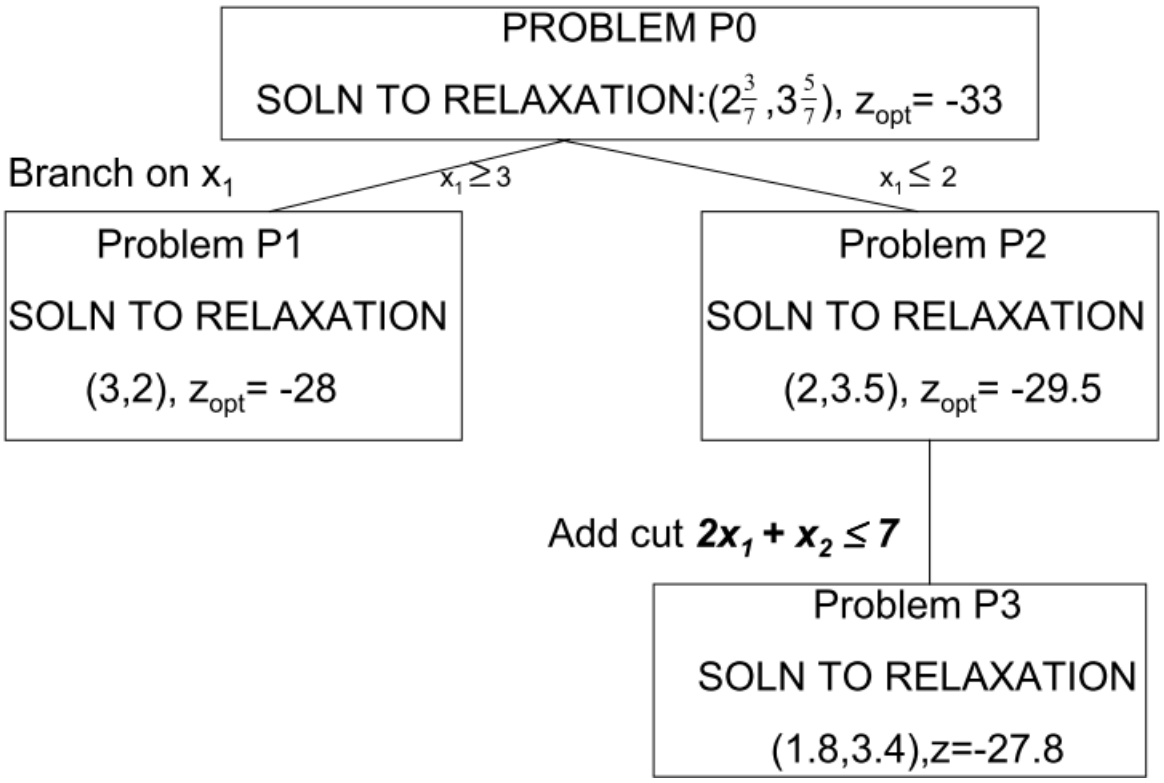


FIGURE 7.6 – Illustration de la procédure Branch and Bound sur le problème (7.7) avec la génération d'une coupe

Plus exactement la procédure Branch and Cut consiste en :

— La procédure est similaire à celle de Branch and Bound.

— Exception faite au niveau de l'étape de la résolution de la relaxation qui sera suivie d'une étape d'ajout de coupes valides.

Il existe plusieurs techniques de génération de coupes. Parmi lesquelles, on peut citer la suivante :

1. Prendre une combinaison linéaire et pondérée des inégalités à partir de la relaxation courante.
2. Exploiter le fait que les variables sont entières, en utilisant le mécanisme d'arrondi.
3. Les coupes générées ainsi, sont appelées les coupes de Chvatal-Gomory.

Par exemple, soit

$$\frac{1}{6}(3x_1 + x_2 \leq 11) + \frac{5}{12}(-x_1 + 2x_2 \leq 5$$
donnant $x_2 \leq 3\frac{11}{12}$

car $x_1 < 12$. La partie gauche doit être entière, d'où l'arrondi de la partie droite, donnant finalement :

$$x_2 \leq 3.$$

Soit l'exemple d'une coupe de Gomory. Soit le problème [2] :

2. http ://mat.gsia.cmu.edu/classes/integer/node15.html

$$\begin{array}{ll} \text{maximize} & z = 7x_1 + 9x_2 \\ \text{subject to} & -x_1 + 3x_2 \leq 6 \\ & 7x_1 + x_2 \leq 35 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \text{variables entières.} \end{array} \tag{7.8}$$

Lors de la résolution de sa relaxation LPR, on obtient la tableau final du simplex donné dans la Figure 7.7

Variable	x_1	x_2	s_1	s_2	$-z$	RHS
x_2	0	1	7/22	1/22	0	7/2
x_1	1	0	-1/22	3/22	0	9/2
$-z$	0	0	28/11	15/11	1	63

FIGURE 7.7 – Tableau final du simplex du problème (7.8)

En considérant la première contrainte du tableau du simplex, on a :

$$x_2 + 7/22s_1 + 1/22s_2 = 7/2$$

qu'on peut réécrire en

$$x_2 - 3 = 1/2 - 7/22s_1 - 1/22s_2$$

La partie gauche est purement entière. La partie droite est composée de deux négatifs additionnés à un positif fractionnaire. D'où le fait que :

$$1/2 - 7/22s_1 - 1/22s_2 \leq 0.$$

Cette coupe est dire "coupe de Gomory".

L'idée des coupes est de se rapprocher le plus de l'idée illustrée dans la Figure 7.8.

http://www.more.ms.unimelb.edu.au/students/operationsresearch/lecturenotes/620362_Gomory.pdf

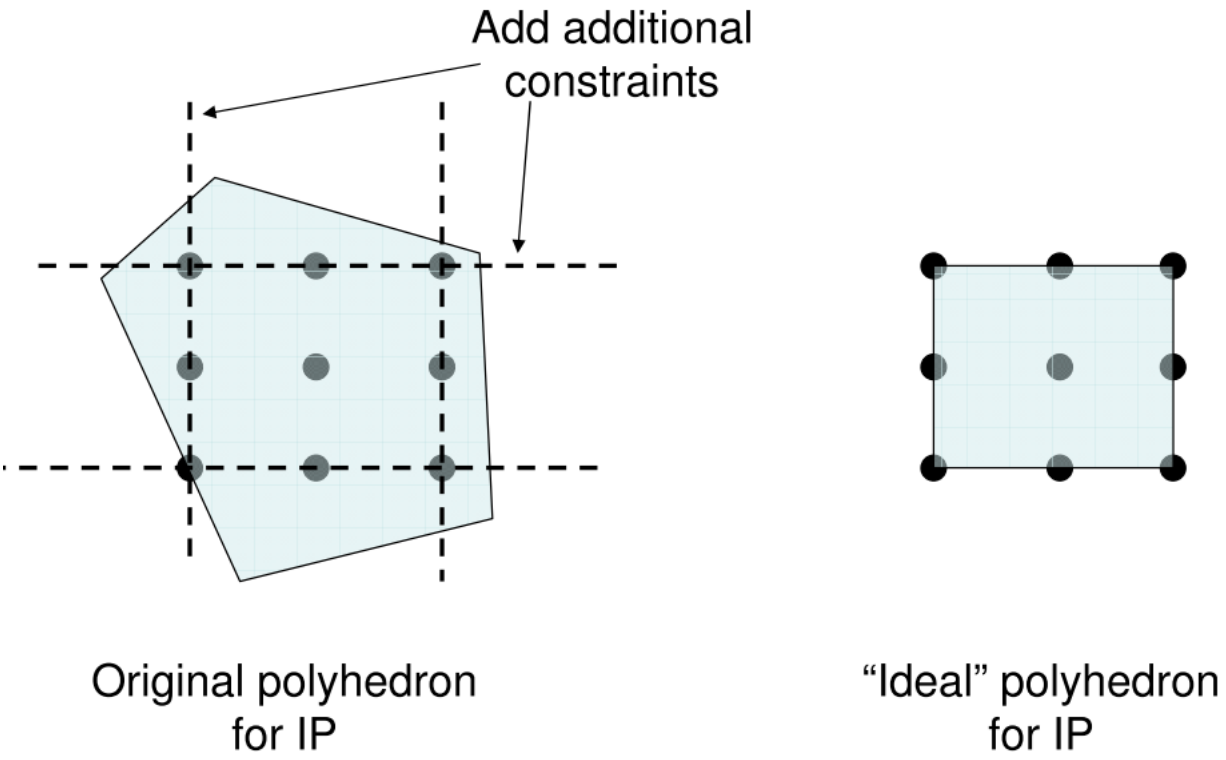


FIGURE 7.8 – Illustration des coupes idéales

Plus généralement, voici ces deux coupes formellement définies.

Proposition 2 (Coupe de Gomory). *Soit* $P = \{x \in \mathbb{Z}_+^n : Ax \leq b\}$. *Soit* $\alpha x \leq \beta$ *valide dans* P *alors*

$$\sum_{i \in 1..n} \lfloor \alpha_i \rfloor x_i \leq \lfloor \beta \rfloor$$

est aussi valide pour P.

Proposition 3 (Coupe de Chvatal-Gomory). *Soit* $P = \{x \in \mathbb{Z}_+^n : Ax \leq b, x \geq 0\}$, *avec* $A \in \mathbb{R}^{m \times n}$ *et* $u \in \mathbb{R}_+^m$, *alors*

$$\sum_{j \in 1..n} \lfloor \sum_{i \in 1..n} u_i A_{i,j} \rfloor x_j \leq \lfloor \sum_{i \in 1..m} u_i b_i \rfloor$$

est aussi valide pour P.

7.3 Correction [Cormen et al., 2001]

On dit qu'un algorithme est correct si, pour chaque instance d'entrée, il se termine avec la sortie désirée correcte. Dans ce document, nous décrirons généralement les algorithmes au moyen de programmes écrits en pseudo-code, très proche de C, Java, ou aussi Pascal.

Nous rappelons que l'invariant d'une boucle, dans un algorithme, est une propriété qui est tout le temps vraie à :

1. l'entrée du corps de la boucle,
2. la sortie du corps de la boucle,
3. à la fin de la boucle.

Pour prouver l'invariant, il est nécessaire de procéder comme suit :

Initialisation La propriété invariante est vraie à l'entrée de la première itération.

Maintenance Si la propriété est vraie dans une itération, l'invariant reste vraie dans l'itération suivante.

Terminaison A la fin de la boucle, l'invariant est suffisant pour démontrer la correction.

L'établissement de cette preuve permet systématiquement d'avoir la preuve de correction de l'algorithme.

Exemple

L'invariant de la boucle de l'algorithme est :

Invariant 1 (Invariant de la boucle TRI-INSERTION)**.** *Le sous-tableau* $A[1..j-1]$ *contient tous les éléments originaux de* $A[1..j-1]$*, et dans un ordre croissant.*

On exploitera cette propriété pour démontrer que l'algorithme est correct.
La preuve se présente comme suit :

Initialisation Il est évident que $A[1..1]$ est bien trié.

Maintenance Supposons que l'algorithme est arrivé à la $j$ième itération, d'où $A[1..j-1]$ est trié. A la fin de la $j$ième itération, l'algorithme aurait placé $A[j]$ à la première cas i, $i < j$ où $A[i] > A[j]$. Ceci va faire en sorte à ce que $A[1..j]$ soit aussi trié.

Terminaison A la fin de la boucle, on aurait $A[1..n]$ trié. D'où la correction de l'algorithme.

7.4 Algorithmique et complexité

Une des premières références historiques sur les algorithmes date du neuvième siècle, à Baghdad, dont l'auteur est Al Khwarizmi qui a proposé les méthodes de base pour additionner, multiplier, et diviser des nombres entiers - et aussi le calcul des racines d'une équation, et les décimales du nombre π. Ces procédures étaient précises, non-ambigües, mécaniques, efficaces, et correctes : elles étaient simplement des *algorithmes*, un terme qui a honoré son concepteur El Khwarizmi.

Depuis cet avènement, et celui de l'adoption du système décimal, les scientifiques ont développé, et développent des algorithmes complexes pour résoudre des problèmes variés.

7.4.1 Exemple de motivation [Papadimitriou et al., 2006]

Soit la suite de Fibonacci (F_n)

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{si n>1,} \\ 1 & \text{si n=1,} \\ 0 & \text{si n=0.} \end{cases}$$

La suite génère les nombres suivants

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...$$

Cette suite a aussi la caractéristique d'être de nature exponentielle, d'ailleurs il existe l'approximation $F_n \approx 2^{0.694n}$. Mais, comment pourrons nous calculer des nombres aussi grands que F_{100} ou même F_{200} ?

7.4.1.1 Une première version

Algorithm 1 FIB1(n)

```
1: if n = 0 then
2:    return 0
3: else if n = 1 then
4:    return 1
5: else
6:    return fib1(n − 1) + fib1(n − 2)
7: end if
```

Trois questions essentielles sont posées :

1. **L'algorithme est-il correct ?**
2. **Combien de temps exige-t-il pour terminer, en fonction de n ?**
3. **Peut-on l'améliorer ?**

La réponse à la première question est plutôt évidente : l'algorithme est bien correct, car il met en oeuvre exactement la définition récurrente de la suite de Fibonacci. La deuxième question est délicate. Soit $T(n)$ le nombre de pas dont a besoin l'algorithme pour calculer F_n. Une première évidence :

$$T(n) \leq 2 \text{ pour } n \leq 1.$$

Pour des nombres n plus grands, on a la relation suivante :

$$T(n) = T(n-1) + T(n-2) + 3 \text{ pour } n > 1.$$

Nous disposons d'une solution approchée $T_n \approx 2^{0.694n}$. Ceci veut dire que le nombre de pas croit aussi vite que la suite de Fibonacci. $T(n)$ a évidemment une **croissance exponentielle**.

Soit par exemple le calcul de F_{200}, le calcul de FIB1(n) exécute $T(200) \geq F_{200} \geq 2^{138}$ pas élémentaires. On se pose la question sur le temps nécessaire pour ce calcul sur un ordinateur ? La réponse est immédiate : prenons un ordinateur puissant à l'heure actuelle, exécutant 40 mille milliards 40×10^{12} instructions par seconde (un ordinateur personnel, un PC, avec un processeur i7-3770 peut exécuter 20×10^{10} opérations flottantes par seconde - on dit 200 gigaFLOPS). Nous aurons besoin sur cet ordinateur de 2^{92} **secondes** ! Ce qui veut dire que l'on doit attendre jusqu'à l'extinction théorique du soleil !

D'après l'hypothèse de Moore qui suppose que les ordinateurs doublent leur puissance tous les 18 mois, alors, on pourrait arriver au fait que : si on arrivait à calculer raisonnablement F_{100}, alors avec l'hypothèse de Moore, on pourrait vraisemblablement, calculer avec la même puissance $F(101)$, ... Voulant dire qu'on gagnera chaque année un chiffre. C'est ainsi qu'on se voit totalement découragé face à la réalité terrible de la croissance exponentielle qui **ne peut être cassée par aucune machine de nos jours !**

La question reste posée : **peut on mieux faire en terme de coût de calcul ?** Et sans passer par l'amélioration des machines, dont on vient de voir l'impuissance !

7.4.1.2 Une version polynomiale

On montre dans la Figure 7.9, le comportement de l'algorithme FIB1.

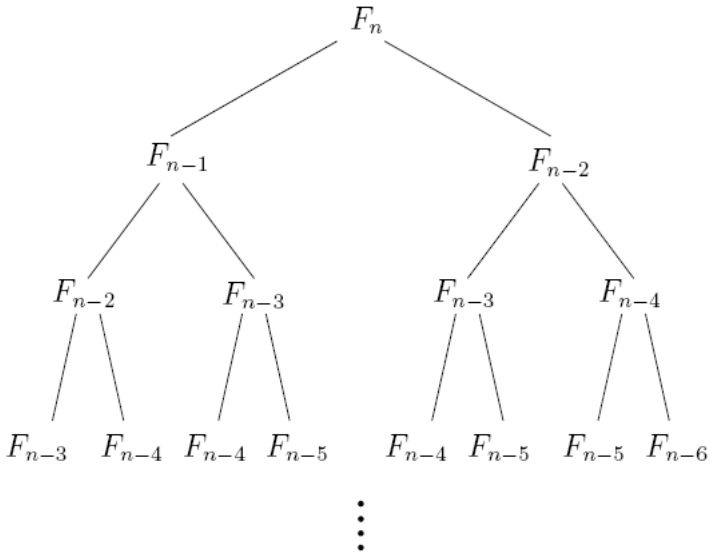


FIGURE 7.9 – Trace de FIB1(n) [Papadimitriou et al., 2006]

En analysant cette trace, on voit bien que plein d'appels sont répétés. On peut donc adopter l'astuce qui consiste à stocker les calculs déjà faits. On obtient la version FIB2.

Algorithm 2 FIB2(n)

```
1: if n = 0 then
2:    return 0
3: end if
4: créer un tableau F[0..n]
5: F[0] ← 0
6: F[1] ← 1
7: for i ← 2..n do
8:    F[i] ← F[i − 1] + F[i − 2]
9: end for
10: return F[n]
```

L'algorithme est évidemment correct, car, encore une fois, il est similaire à la définition récurrente de Fibonacci. La question : combien requiert cet algorithme pour calculer F_n en fonction de n ? La boucle de l'algorithme nécessite $n-1$ pas de calculs. De ce fait, le temps d'exécution de **cet algorithme est de nature linéaire en** n. Il devient maintenant possible de calculer F_{200000} **avec peu d'efforts !**

Bien concevoir un algorithme fait gagner des exponentielles !

7.4.2 Complexité [Cormen et al., 2001]

Un algorithme est un ensemble d'opérations de calcul élémentaires, organisées selon des règles précises dans le but de résoudre un problème donné. Pour chaque donnée du problème, l'algorithme retourne une réponse après un nombre fini d'opérations. Les opérations élémentaires sont par exemple les opérations arithmétiques usuelles, les transferts de données, les comparaisons entre données, etc.

Il apparaît utile de ne **considérer comme véritablement élémentaires que les opérations dont le temps de calcul est constant**, c'est-à-dire ne dépend pas de la taille des opérandes. Par exemple :

— l'addition d'entiers de taille bornée à priori (les int en C) est une opération élémentaire ;
— l'addition d'entiers de taille quelconque ne l'est pas.
— De même, le test d'appartenance d'un élément à un ensemble n'est pas une opération élémentaire en ce sens, parce que son temps d'exécution dépend de la taille de l'ensemble, et ceci même si dans certains langages de programmation, il existe des instructions de base qui permettent de réaliser cette opération.

Nous avons déjà pris goût aux algorithmes avec l'algorithme de calcul des termes de la suite de Fibonacci pour lequel nous avons introduit une première analyse simple. Pour illustrer notre démarche, nous aborderons un problème posé fréquemment en pratique, celui du tri d'une suite de nombres en ordre croissant.

Nous pouvons définir formellement ce problème comme suit :

Entrée Une séquence de n nombres $\langle a_1, a_2, ..., a_n \rangle$.

Sortie Une permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ de la permutation de la suite d'entrée telle que $a_1 \leq a_2 \leq ... \leq a_n$.

Etant donnée une suite d'entrée telle que $\langle 5, 90, 45, 89 \rangle$, un algorithme de tri fournira en sortie $\langle 5, 45, 89, 90 \rangle$. On dira d'une telle suite d'entrée que c'est une instance du problème.

Nous introduirons une première solution au problème du tri avec la version du TRI PAR INSERTION.

Algorithm 3 TRI-INSERTION(A)

Input: Une séquence de n nombres $A = \langle a_1, a_2, ..., a_n \rangle$.

Output: Une permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ de la permutation de la suite d'entrée telle que $a_1 \leq a_2 \leq ... \leq a_n$.

```
1: for j ← 2 to longueur(A) do
2:    pivot ← A[j]
3:    {insertion de A[j] dans la suite triée A[1..j − 1]}
4:    i ← j − 1
5:    while (i > 0) ∧ (A[i] > pivot) do
6:       A[i + 1] ← A[i]
7:       i ← i − 1
8:    end while
9:    A[i + 1] ← pivot
10: end for
```

Le temps pris par la procédure TRI-INSERTION dépend de la **taille de son entrée** : le tri d'un millier de nombres prend plus de temps que le tri de trois nombres. La notion de taille d'entrée dépend du problème étudié, parfois il est plus approprié de décrire la taille de l'entrée avec deux nombres. Par exemple, la taille de l'entrée dans un problème sur les graphes, peut être le nombre de sommets et le nombres d'arêtes. **Pour chaque problème étudié, nous préciserons la taille (mesure) utilisée.**

Le temps d'exécution d'un algorithme sur une entrée particulière est le nombre d'opérations élémentaires exécutées.

Le temps d'exécution de l'algorithme est la somme des temps d'exécution de chaque instruction exécutée ; une instruction qui s'exécute en un temps c_i et qui est exécutée n fois interviendra pour $c_i n$. Pour calculer $T(n)$, le temps d'exécution de TRI-INSERTION, on additionne les produits des coûts par le nombre de fois, et on obtient :

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2..n} t_j + c_6 \sum_{j=2..n} (t_j - 1) + c_7 \sum_{j=2..n} (t_j - 1) + c_9(n-1),$$

où t_j est le nombre d'itérations de la deuxième boucle à l'itération j de la première boucle. Si la suite est déjà ordonnée, on a

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) = (c_1 + c_2 + c_4 + c_5 + c_9)n + (c_2 + c_4 + c_5 + c_9).$$

On peut exprimer cette dernière formule sous la forme $an + b$, où a et b sont des constantes. On dit que l'algorithme est en $O(n)$.

Si la suite est dans un ordre inverse, on se trouve dans le pire des cas. On doit comparer chaque élément $A[j]$ avec chaque élément de sous-tableau trié $A[1..j-1]$, et donc $t_j = j$ pour $j = 2, 3, ..., n$. Sachant que

$$\sum_{j=2..n} j = n(n+1)/2 - 1$$

et

$$\sum_{j=2..n} (j-1) = n(n-1)/2.$$

On obtient ainsi

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2-1) + c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_9(n-1)$$

$$T(n) = (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 + c_6/2 + c_7/2 + c_9)n.$$

Ce dernier coût est de la forme an^2+bn+c où a, b et c sont des constantes. On dit que l'algorithme est en $O(n^2)$. Nous venons donc de calculer le coût de l'algorithme du TRI-INSERTION dans le pire des cas, c'est-à-dire le temps d'exécution le plus long pour une entrée quelconque de taille n. C'est la mesure de coût la plus utilisée en pratique. Nous verrons dans la suite un ensemble de procédés permettant de calculer ce coût.

7.5 Mesure de complexité

Analyser un algorithme consiste à prévoir **les ressources à cet algorithme**. Parfois, les ressources pertinentes sont la **mémoire** utilisée, la **largeur de bande d'une communication**, ou les **portes logiques**, mais le plus souvent, on souhaite mesurer le **temps de calcul**. Dans ce cours, on prendra comme modèle de calcul, **une machine à accès aléatoire, à processeur unique**. Dans ce modèle, les instructions sont exécutées l'une après l'autre, sans opérations simultanées.

Considérons un problème donné, et un algorithme pour le résoudre. Sur une donnée x de taille n, l'algorithme requiert un certain temps, mesuré en nombre d'opérations élémentaires, soit $c(x)$. Le coût en temps varie évidemment avec la taille de la donnée, mais peut aussi varier sur les différentes données de même taille n.

Par exemple, considérons l'algorithme de tri qui, partant d'une suite $(a_1, ..., a_n)$ de nombres réels distincts à trier en ordre croissant, cherche la première descente, c'est-à-dire le plus petit entier i tel que $a_i > a_{i+1}$, échange ces deux éléments, et recommence sur la suite obtenue. Si l'on compte le nombre d'inversions ainsi réalisées, il varie de 0 pour une suite triée à $n(n-1)/2$ pour une suite décroissante. Notre but est d'**évaluer le coût d'un algorithme, selon certains critères, et en fonction de la taille n des données**.

Pour certains problèmes, on peut **mettre en évidence une ou plusieurs opérations qui sont fondamentales** au sens où le temps d'exécution d'un algorithme résolvant ce problème est toujours proportionnel au nombre de ces opérations. Il est alors possible de comparer des algorithmes traitant ce problème selon cette mesure simplifiée.

Donnons quelques exemples d'**opérations fondamentales** :

1. pour la recherche d'un élément dans une liste en mémoire centrale : le nombre de comparaisons entre cet élément et les entrées de la liste ;
2. pour la recherche d'un élément sur un disque : le nombre d'accès à la mémoire secondaire ;
3. pour trier une liste d'éléments : on peut considérer deux opérations fondamentales : le nombre de comparaisons entre des éléments et le nombre de déplacements d'éléments ;
4. pour multiplier deux matrices de nombres : le nombre de multiplications et le nombre d'additions.

Remarquons que si l'on choisit plusieurs opérations fondamentales, on **peut les décompter séparément** puis, si besoin est, on les affecte chacune d'un poids qui tient compte des temps d'exécution différents.

1. En faisant **varier le nombre d'opérations fondamentales**, on fait varier le **degré de précision de l'analyse**, et aussi son degré d'abstraction, i.e. d'indépendance par rapport à l'implémentation. A la limite, si l'on veut **faire une microanalyse très précise du temps d'exécution du programme, il suffit de décider que toutes les opérations du programme sont fondamentales**.
2. On a fait l'hypothèse que le temps d'exécution est proportionnel à la mesure choisie. On ne peut pas comparer deux algorithmes utilisant des mesures différentes.

Après avoir déterminé les opérations fondamentales, il s'agit de compter le nombre d'opérations de chaque type. Il n'existe pas de systèmes complet de règles permettant de compter le nombre d'opérations en fonction de la syntaxe des algorithmes mais l'on peut faire **quelques remarques** :

Séquence Lorsque les opérations sont dans une séquence d'instructions, leurs nombres s'ajoutent.

if-then-else Pour les branchements conditionnels, il est en général difficile de déterminer quelle branche de la condition est exécutée, et donc quelles sont les opérations à compter. Cependant, on peut majorer ce nombre d'opérations.

Boucles Pour les boucles, le nombre d'opérations dans la boucle est $\sum P(i)$, où i est la variable de contrôle de la boucle, et $P(i)$ le nombre d'opérations fondamentales lors de l'exécution de la ième itération. Si le nombre d'itérations est difficile à calculer, on peut se contenter d'une bonne majoration.

Appels procédures Pour les appels de procédures et fonctions non récursives, on peut s'arranger à calculer la complexité de ces appels, et les prendre en compte suivant l'imbrication de l'appel dans l'algorithme.

Appels récursifs Pour les appels de procédures et fonctions récursives, compter le nombre d'opérations fondamentales donne en général lieu à la **résolution de relations de récurrence**. En effet le nombre $T(n)$ d'opérations dans l'appel de la procédure avec un argument de taille n s'écrit, selon la récursion, en **fonction de divers $T(k)$, pour $k < n$. L'exemple de Fibonacci** donné dans le chapitre d'introduction illustre bien ce cas.

Il est évident que **le calcul du coût d'un algorithme dépend de la donnée sur laquelle il opère**.

1. Il faut d'abord **définir une mesure de taille sur les données** qui reflète la quantité d'information contenue. **Par exemple, si l'on additionne ou on multiplie des entiers, une mesure significative est le nombre de chiffres des nombres**.
2. Pour certains algorithmes, le temps d'exécution ne dépend que de la taille des données ; mais la plupart du temps la complexité varie aussi, pour une taille fixée des données, en fonction de la donnée elle-même.

7.5.1 La complexité dans le meilleur des cas

Le coût $Min_A(n)$ d'un algorithme A dans le meilleur des cas

$$Min_A(n) = min_{|x|=n} c(x).$$

7.5.2 La complexité dans le pire des cas

Le coût $Max_A(n)$ d'un algorithme A dans le cas le plus défavorable ou dans le cas le pire [3] est par définition le maximum des coûts, sur toutes les données de taille n :

$$Max_A(n) = max_{|x|=n} c(x).$$

7.5.3 La complexité en moyenne

Dans des situations où l'on pense que **le cas le plus défavorable ne se présente que rarement, on est plutôt intéressé par le coût moyen de l'algorithme**. Une formulation correcte de ce coût moyen suppose que l'on connaisse une distribution de probabilités sur les données de taille n. **Si $p(x)$ est la probabilité de la donnée x**, le coût moyen $Moy_A(n)$ d'un algorithme A sur les données de taille n est par définition

$$Moy_A(n) = \sum_{|x|=n} p(x)c(x).$$

Le plus souvent, on suppose que la distribution est uniforme, c'est-à-dire que $p(x) = 1/T(n)$, où $T(n)$ est le nombre de données de taille n. Alors, l'expression du coût moyen prend la forme

$$Moy_A(n) = \frac{1}{T(n)} \sum_{|x|=n} c(x).$$

En pratique, la complexité en moyenne est souvent beaucoup plus difficile à déterminer que la complexité dans le pire des cas , d'une part parceque l'analyse devient mathématiquement difficile, et d'autre part parcequ'il n'est pas toujours facile de déterminer un modèle de probabilités adéquat au problème.

En claire, il existe entre les complexités en moyenne et les complexités extrêmales la relation suivante :

$$Min_A(n) \leq Moy_A(n) \leq Max_A(n).$$

Si le comportement de l'algorithme dépend uniquement de la taille des données (comme dans l'exemple de la multiplication de matrices), alors ces trois quantités sont confondues. Mais en général, ce n'est pas le cas et l'on ne sait même pas si le coût moyen est plus proche du coût minimal ou du coût maximal.

3. "worst-case" en anglais.

7.5.4 Grandeurs des fonctions et notations de Landau : O, ω, ... [Gaudel et al., 1990]

On a déterminé **la complexité d'un algorithme comme une fonction de la taille des données** ; il est très important de connaitre la **rapidité de croissance** de cette fonction lorsque la **taille des données croît**. En effet, pour traiter un problème de petite taille la méthode employée importe peu, alors que pour un problème de grande taille, les différences de performance entre algorithmes peuvent être énormes.

Souvent une simple approximation de la fonction de complexité suffit pour savoir si un algorithme est utilisable ou non, ou pour **comparer entre différents algorithmes**.

Par exemple, pour n grand, il est souvent secondaire de savoir si un algorithme fait $n+1$ ou $n+2$ opérations.

Parfois les constantes multiplicatives ont, elles aussi, peu d'importance.

- Supposons que l'on ait à comparer l'algorithme A_1 de complexité $M_1(n) = n^2$ et l'algorithme A_2 de complexité $M_2(n) = 2n$. **A_2 est meilleur que A_1** pour presque tous les $n(n > 2)$;
- De même si $M_1(n) = 3n^2$ et $M_2(n) = 25n$; **A_2 est meilleur que A_1** pour $n > 8$.
- **[*Négliger un terme dans une addition*] Quelles que soient les constantes multiplicatives k_1 et k_2 telles que $M_1(n) = k_1 n^2$ et $M_2(n) = k_2 n$, l'algorithme A_2 est toujours meilleur que A_1 à partir d'un certain n, car la fonction $f(n) = n^2$ croît beaucoup plus vite que la fonction $g(n) = n$. En effet**

$$\lim_{n\to\infty} g(n)/f(n) = 0.$$

  On dit que l'ordre de grandeur asymptotique de $f(n)$ est strictement plus grand que celui de $g(n)$.

- **[*Négliger une constante*] D'autre part, quelles que soient les constantes multiplicatives k_1 et k_2 telles que $M_1(n) = k_1 f(n)$ et $M_2(n) = k_2 f(n)$, l'algorithme A_2 est toujours du meme ordre que A_1, en effet**

$$\lim_{n\to\infty} k_1 f(n)/k_2 f(n) = k_1/k_2.$$

La Figure 7.10 met en évidence la différence de rapidité de croissance de certaines fonctions usuelles : les ordres de grandeur asymptotique des fonctions $1, log_2(n), nlog_2(n), n^2, n^3, 2^n$ vont en croissant strictement ; ces fonctions forment une échelle de comparaison.

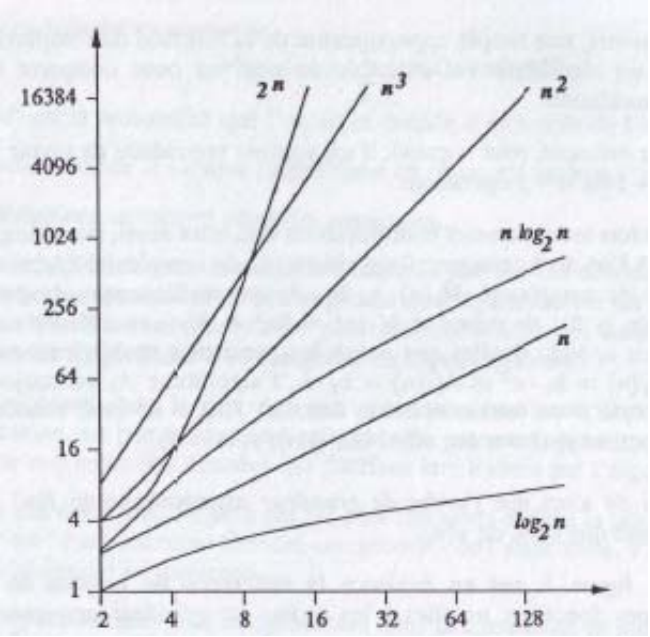


FIGURE 7.10 – Croissance de certaines fonctions usuelles [Gaudel et al., 1990]

Pour analyser la complexité $M_A(n)$ d'un algorithme A, on s'attache d'abord à déterminer l'ordre de grandeur asymptotique de $M_A(n)$: **on cherche dans une échelle de comparaison, éventuellement plus complète que celle qui est formée par les fonctions de la Figure 7.10, une fonction qui a une rapidité de croissance voisine de $M_A(n)$.**

Supposons que l'on ait à comparer deux algorithmes A_1 et A_2 de complexités $M_{A_1}(n)$ et $M_{A_2}(n)$. **Si l'ordre de grandeur de $M_{A_1}(n)$ est strictement plus grand que l'ordre de grandeur de $M_{A_2}(n)$, alors on peut conclure immédiatement que A_1 est meilleure que A_2 pour n grand.** Par contre, si $M_{A_1}(n)$ et $M_{A_2}(n)$ ont même ordre de grandeur asymptotique, il faut faire une analyse plus fine pour pouvoir comparer A_1 et A_2.

Pour comparer l'odre de grandeur asymptotique des fonctions, on a l'habitude d'utiliser la notion suivante :

Définition 13. *Etant donné deux fonctions f et g de $\mathbb{N}$ dans $\mathbb{R}^+$,*

$$f = O(g)$$

si et seulement si $\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}$ tel que

$$\forall n > n_0, f(n) \leq c.g(n).$$

On dit aussi que

$$g = \Omega(f)$$

si et seulement si $f = 0(g)$.

Ainsi $f = O(g)$ **veut dire que l'ordre de grandeur asymptotique de f, est inférieur ou égal à celui de g**, on dit que f **est dominée asymptotiquement par g**; par exemple $2n = O(n^2)$, mais aussi $2n = O(n)$.

Cette notion qui donne un majorant de l'ordre de grandeur asymptotique de f, est très utile pour de nombreuses applications, **mais elle n'est pas suffisante pour comparer entre elles les performances de différents algorithmes**, car il faut connaitre les ordres de grandeurs exacts, et non des majorants. Lorsque l'on dit que la complexité $M_A(n)$ d'un algorithme A est en $h(n)$, on veut dire que son ordre de grandeur asymptotique est exactement $h(n)$ (i.e. $h(n)$ est le plus petit majorant).

On est donc amené à introduire la définition suivante :

Définition 14. *Etant donné deux fonctions f et g de $\mathbb{N}$ dans $\mathbb{R}^+$,*

$$f = \Theta(g)$$

si et seulement si $\exists c, d \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}$ tel que

$$\forall n > n_0, d.g(n) \leq f(n) \leq c.g(n).$$

Ou d'une façon équivalente,

$$f = \Theta(g)$$

si et seulement si $f = 0(g)$ et $g = O(f)$.

On dit que f **et** g **ont même ordre de grandeur asymptotique**. **La notion Θ est plus précise que la notion O**. Par exemple $2n = \Theta(n)$, mais $2n$ **n'est pas en** $\Theta(n^2)$. Cependant, dans la plupart des ouvrages d'algorithmique, les résultats des analyses sont mis sous la forme $O(f(n))$, alors qu'un décompte précis des opérations fondamentales permet souvent de conclure que la complexité est exactement $\Theta(f(n))$. Par exemple, on dit souvent que le tri par tas est en $O(nlog(n))$, alors qu'en fait il est en $\Theta(nlog(n))$.

Soulignons un point fondamental : les définitions de O et Θ reposent sur l'existence de certaines constantes finies, mais il n'est rien précisé sur la valeur de ces constantes. **Cela n'a pas d'importance pour obtenir des résultats asymptotiques lorsque les fonctions ont des ordres de grandeur différents**. Par exemple si $f(n) = 2n$ et $g(n) = n^2$, alors $f(n) < g(n)$ pour $n > 2$. Si $f(n) = 1000n$ et $g(n) = n^2$, alors $f(n) < g(n)$ **pour** $n > 10^4$.

Ainsi, si l'ordre de grandeur de f est plus petit que celui de g alors il existe un seuil à partir duquel la valeur de f est c fois plus petite que celle de g, mais on ne sait pas quel est ce seuil.

Par contre, si f et g ont même ordre de grandeur, il devient beaucoup plus difficile de les comparer : la détermination des constantes, et éventuellement des termes d'ordre inférieur nécessite en général des techniques mathématiques beaucoup plus complexes. Il faut bien être conscient de ce que l'obtention de résultats tels que : "l'algorithme A va deux fois plus vite que l'algorithme B sur un ordinateur standard", est en général très difficile.

La notion d'ordre de grandeur de la complexité des algorithmes a une grande importance pratique. Supposons que l'on dispose pour résoudre un problème donné de sept algorithmes dont les complexités dans le cas le pire ont respectivement pour ordre de grandeur 1 (c'est-à-dire une fonction constante, qui ne dépend pas de la taille des données), $log_2(n)$ (c'est-à-dire une fonction polynomiale d'ordre 3), 2^n (c'est-à-dire une fonction exponentielle).

Complexité / Taille	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n = 10^2$	$\simeq 1\mu s$	6,6 μs	0,1 ms	0,6 ms	10 ms	1 s	4×10^{16} a
$n = 10^3$	$\simeq 1\mu s$	9,9 μs	1 ms	9,9 ms	1 s	16,6 mn	∞
$n = 10^4$	$\simeq 1\mu s$	13,3 μs	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^5$	$\simeq 1\mu s$	16,6 μs	0,1 s	1,6 s	2,7 h	31,7 a	∞
$n = 10^6$	$\simeq 1\mu s$	19,9 μs	1 s	19,9 s	11,5 j	$31,7 \times 10^3$ a	∞

FIGURE 7.11 – Temps d'exécution et taille des données [Gaudel et al., 1990]

Le tableau de la Figure 7.11 donne une estimation du temps d'exéction de chacun de ces algorithmes pour différentes tailles n des données du problème sur un ordinateur pouvant effectuer 10^6 opérations par seconde. **Il montre bien que, plus la taille des données est grande, plus les écarts entre les différents temps d'exécution se creusent.**

Complexité / Temps calcul	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1s	∞	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 jour	∞	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

FIGURE 7.12 – Temps d'exécution et taille des données [Gaudel et al., 1990]

Le tableau de la Figure 7.12 donne une estimation de la taille maximale des données que l'on peut traiter par chacun des algorithmes en un temps d'exécution fixé (et toujours sur un ordinateur effectuant 0^6 opérations par seconde).

D'après ces deux tableaux, il est clair que certains algorithmes sont utilisables pour résoudre des problèmes sur ordinateurs, et que d'autres ne sont pas, ou peu utilisables.

Les algorithmes utilisables pour des données de grande taille sont ceux qui s'exécutent en temps :

constant (c'est le cas de la complexité en moyenne de certaines méthodes de hachage) ;

logatithmique (par exemple la recherche dichotomique, ou les arbres binaires de recherche) ;

linéaire (par exemple la recherche séquentielle) ;

$n.log(n)$ (par exemple les bons algorithmes de tri).

Les algorithmes qui prennent un temps polynomial, c'est-à-dire en $\Theta(n^k)$ avec $k > 0$, ne sont vraiment utilisables que pour $k < 2$.

Lorsque $2 \leq k \leq 3$, on peut traiter que des problèmes de taille moyenne, et lorsque k dépasse 3 on ne peut traiter que des petits problèmes.

Les algorithmes en temps exponentiel, c'est-à-dire en $\Theta(2^n)$ par exemple, sont à peu près inutilisables, sauf pour des problèmes de très petit taille. Ce sont de tels algorithmes que l'on a qualifiés d'inefficaces.

Le tableau de la Figure 7.12 montre comment la taille des données et le temps d'exécution varient en fonction l'un de l'autre. On voit en particulier que si l'on multiplie par 10 la vitesse de calcul de l'ordinateur, on ne modifie quasiment pas la taille maximale des données que l'on peut traiter avec un algorithme exponentiel, alors que l'on multiplie évidemment par 10 la taille des données traitables par un algorithme linéaire. **Il est donc toujours d'actualité de rechercher des algorithmes efficaces, même si les projets technologiques accroissent les performances du matériel.**

Complexité	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
Evolution du temps quand la taille est multipliée par 10	t	t+3,32	10×t	(10+ε)×t	100×t	1000×t	t^{10}
Evolution de la taille quand le temps est multiplé par 10	∞	n^{10}	10×n	(10−ε)×n	3,16×n	2,15×n	n+3,32

FIGURE 7.13 – Temps d'exécution et taille des données [Gaudel et al., 1990]

Taille → / Complexité ↓	20	50	100	200	500	1 000
$10^3.n$	0.02 s	0.05 s	0.1 s	0.2 s	0.5 s	1 s
$10^3.n.\log_2 n$	0.09 s	0.3 s	0.6 s	1.5 s	4.5 s	10 s
$100.n^2$	0.04 s	0.25 s	1 s	4 s	25 s	2 mn
$10.n^3$	0.02 s	1 s	10 s	1 mn	21 mn	27 h
$n^{\log_2 n}$	0.4 s	1.1 h	220 jours	12 500 ans	5.10^{10} ans	--
$2^{n/3}$	0.0001 s	0.1 s	2.7 h	3.10^6 ans	--	--
2^n	1 s	36 ans	--	--	--	--
3^n	58 mn	2.10^{11} ans	--	--	--	--
$n!$	77 100 ans	--	--	--	--	--

FIGURE 7.14 – Temps d'exécution et taille des données [Prins, 1994]

7.6 Nombre et arithmétique des intervalles

Les nombres naturels, $\mathbb{N}$, forment l'ensemble de base qui est la pierre angulaire de la définition des autres structures de nombres. L'ensemble des nombres relatifs est noté $\mathbb{Z}$. L'ensemble des nombres rationnels, Q, est l'ensemble des fractions, ou de couples d'entiers, $Q = \{\frac{x}{y} | x \in \mathbb{Z} \wedge y \in \mathbb{Z}^*\}$. L'ensemble des nombres réels, $\mathbb{R}$, est une extension des nombres rationnels. Un nombre réel est défini comme la limite d'une suite infinie de nombre rationnels. L'ensemble des nombres réels qui ne peuvent pas être mis sous forme rationnelle sont appelés nombres irrationnels, on les notera $\overline{Q}$. Les nombres complexes, C, sont un ensemble de couples de nombres réels. La propriété fondamentale des nombres complexes est que toute équation polynomiale, à coefficients complexes, a des solutions. On dispose dans chacun des ensembles "$\mathbb{N}$, Q, $\mathbb{R}$, C" d'un ensemble de fonctions mathématiques pour faire des calculs.

Les nombres flottants : l'ensemble des réels est infini. En pratique, on dispose de moyens finis pour représenter les nombres et les calculs. Ce qui a donné lieu à l'ensemble des nombres flottants qui est un ensemble fini pour approximer les réels. Un flottant est sous la forme $a \times b^c$, ou a, b et c appartiennent à un sous-domaine fini de $\mathbb{Z}$. Tout nombre d'un système particulier de nombres flottants est spécifié avec une base b fixe[4]. La norme internationale IEEE754 [?] représente le flottant sur 64 bits, $b = 2$, a est stocké sur 53 bits (un bit est utilisé pour représenter le signe) et c sur 11 bits (avec la représentation binaire biaisée). Pour avoir plus d'informations sur un système de flottants, [?] utilise la notation $\mathbb{F}[b, a, n \ldots M]$, avec [n,M] l'intervalle des nombres représentables dans le système flottant utilisé. La norme [?] est notée $\mathbb{F}[2, 53, -2^{10} + 2 \ldots 2^{10} - 1]$. Tout réel ne peut pas être représenté exactement dans le système flottant, et de même pour le calcul sur les flottants, car une opération sur les flottants peut donner un nombre qui n'est pas représentable dans le système flottant utilisé. En conséquence, pour représenter l'approximation d'un nombre réel

4. La base b des systèmes flottants est souvent la base binaire.

x, tout système flottant[5] fournit les quatre routines suivantes :

1. *roundnear* : impose au système de prendre le flottant le plus proche du nombre x ;
2. *roundup* : impose au système de prendre le nombre le plus petit parmi les flottants qui sont plus grands que le nombre x ;
3. *rounddown* : impose au système de prendre le nombre le plus grand parmi les flottants qui sont plus petits que le nombre x.
4. *roundzero* : impose au système de prendre parmi les deux flottants qui encadrent x celui qui est le plus proche de zéro.

Les machines actuelles utilisent l'arithmétique en virgule flottante. Dans cette arithmétique, les nombres réels sont approximés par un sous-ensemble fini de réels appelés nombres machine. À cause de la finitude de cet ensemble, deux erreurs sont générées immanquablement. La première erreur est produite lors de l'approximation d'un nombre réel par un nombre flottant. La deuxième est générée par les calculs intermédiaires réels approximés sur les nombres flottants. En pratique, on peut tomber sur des calculs qui donnent des résultats qui sont éloignés de la bonne valeur.

Exemple 3. *[?]*
Soit l'expression

$$333.75 \times y^6 + x^2 \times (11 \times x^2 \times y^2 - y^6 - 121 \times y^4 - 2) + 5.5 \times y^8 + \frac{x}{2 \times y}.$$

L'évaluation de cette expression au point $(x = 77617, y = 33096)$ *donne avec la représentation double* $1.17260\ldots$*, avec la représentation étendue* $1.17260\ldots$*, alors que la valeur exacte est* $-0.82739\ldots$*!*

D'ailleurs, en analyse numérique, les algorithmes sont souvent présentés avec une étude de stabilité numérique pour caractériser la sensibilité de l'algorithme aux erreurs d'approximation.

La raison principale de l'instabilité numérique, en utilisant l'arithmétique standard, est que l'arithmétique flottante ne respecte pas les propriétés de l'arithmétique réelle. L'arithmétique des intervalles donne les moyens de calcul pour estimer et contrôler automatiquement les erreurs d'approximation. Contrairement à l'arithmétique standard, l'arithmétique flottante des intervalles respecte les propriétés de l'arithmétique réelle des intervalles.

L'arithmétique des intervalles a été introduite par [?]. Nous présenterons brièvement cette arithmétique en montrant ses avantages par rapport à l'arithmétique standard, dans le même ordre d'idée que [?].

Dans l'arithmétique réelle des intervalles, un nombre réel e est approximé par deux nombres réels l et r tel que $l \leq e \leq r$. Le nombre e est donc représenté par l'intervalle $[l, r] = \{x | l \leq x \leq r\}$. La taille de cet intervalle donne une mesure qualitative de l'approximation. Les calculs sont faits sur les intervalles avec la garantie d'encadrement des résultats intermédiaires.

5. [?] présente une bonne synthèse des propriétés des systèmes flottants.

Soit E un ensemble partiellement ordonné par la relation $\leq$, on note par $I\!I(E) = \{[x, y] | x \in E, y \in E, x \leq y\}$ l'ensemble des intervalles construits sur E. L'intervalle contenant tous les éléments de l'ensemble $Y \subseteq E$, est dénoté $\Box Y$. Si $\min(Y)$ et $\max(Y)$ existent, alors $\Box Y = [\min(Y), \max(Y)]$. La borne inférieure et la borne supérieure d'un intervalle X sont respectivement dénotées par $\underline{X}$ et $\overline{X}$.

$I\!I(I\!R)$ est l'ensemble des intervalles sur $I\!R$. $wid(X) = \overline{X} - \underline{X}$ est la largeur de l'intervalle X. La largeur d'un vecteur d'intervalles X est

$$wid(X) = max(wid(X_1), \ldots, wid(X_n)).$$

Nous définissons de la même façon la largeur d'une matrice d'intervalles.

$mid(X) = (\overline{X} - \underline{X})/2$ est le point milieu de l'intervalle X. Le milieu d'un vecteur d'intervalles X est

$$mid(X) = \langle mid(X_1), \ldots, mid(X_n) \rangle.$$

Nous définissons de la même façon le milieu d'une matrice d'intervalles.

La distance $dist(X, Y)$, avec $X \in I\!I(I\!R), Y \in I\!I(I\!R)$, (resp. $X \in I\!I(I\!R)^n, Y \in I\!I(I\!R)^n$) est définie par $\max(|\underline{X} - \underline{Y}|, |\overline{X} - \overline{Y}|)$ (resp.

$$\max\{dist(X_1, Y_1), \ldots, dist(X_n, Y_n)\}).$$

De la même façon, on définit la distance entre deux matrice.

On peut démontrer [?] que cette distance donne lieu à une topologie d'espace métrique $(I\!I(I\!R), dist)$. On trouve dans [?], une définition de la continuité, de la continuité uniforme et de propriétés analytiques sur les intervalles.

Soient A et B des nombres intervalles de $I\!I(I\!R)$. Les quatre opérations $\{\times, +, -, /\}$ sur $I\!I(I\!R)$ doivent satisfaire :

$$A \; op \; B \supseteq \{a \; op \; b | a \in A, b \in B\}, \text{ avec } op \in \{\times, +, -, /\}$$

Nous garderons la même propriété sur l'extension des autres opérations unaires, trigonométriques, etc.

La multiplication, l'addition, la soustraction et la division sont définies dans (7.9). [?, ?, ?, ?] ont proposé plusieurs améliorations pour avoir des bornes plus précises du calcul de la division par un intervalle contenant *zéro*.

$$\begin{array}{rcl} [a,b] + [c,d] & = & [a + c, b + d] \\ [a,b] - [c,d] & = & [a - d, b - c] \\ [a,b] \times [c,d] & = & \\ & & [\\ & & \quad \min(a \times c, a \times d, b \times c, b \times d), \\ & & \quad \max(a \times c, a \times d, b \times c, b \times d) \\ & &] \\ [a,b]/[c,d] & = & [a,b] \times [1/d, 1/c] \textbf{ if } 0 \notin [c,d] \\ & & [-\infty, +\infty] \textbf{ if } 0 \in [c,d] \end{array} \tag{7.9}$$

La soustraction (resp. la division) n'est pas l'opération inverse de l'addition (resp. multiplication). La plupart des propriétés de distributivité de l'arithmétique réelle sont absentes de l'arithmétique des intervalles. [?] a démontré que les quatre opérations arithmétiques sur les intervalles définies par (7.9) sont continues, sauf la

division par des intervalles contenant *zéro*. Les fonctions rationnelles sur les intervalles sont aussi continues.

L'ensemble $I\!I(I\!R)$ est muni de la relation binaire $\leq$ définie par :

$$\begin{array}{l} X \in I\!I(R), Y \in I\!I(R), X \leq Y \equiv \overline{X} \leq \underline{Y} \\ X \in I\!I(R)^n, Y \in I\!I(R)^n, X \leq Y \equiv (\overline{X_i} \leq \underline{Y_i}, i = 1 \ldots n) \\ X \in I\!I(R)^{m \times n}, Y \in I\!I(R)^{m \times n}, \\ \quad X \leq Y \equiv (\overline{X_{i,j}} \leq \underline{Y_{i,j}}, i = 1 \ldots m, j = 1 \ldots n) \end{array} \tag{7.10}$$

L'ensemble des intervalles (resp. vecteurs d'intervalles et matrices d'intervalles) munis de la relation (7.10) a une structure de treillis.

En pratique, nous utiliserons l'arithmétique *flottante* des intervalles qui est une approximation discrète de l'arithmétique réelle des intervalles. Soient α et β deux inconnues réelles d'un calcul intermédiaire dans un calcul donné. Nous pouvons encadrer les deux inconnues $\alpha \in A$ et $\beta \in B$ par deux intervalles réels A et B. Les deux bornes des deux intervalles A et B sont des réels, donc ne peuvent être représentées sur machine. L'idée est de les représenter par les deux plus petits intervalles flottants [6] qui contiennent ces deux intervalles, soit A_f et B_f avec $A \subseteq A_f$ et $B \subseteq B_f$. Pour toute opération *op* de l'arithmétique réelle nous disposons de son équivalent en arithmétique réelle des intervalles, qui garantit que : $(A\ op\ B) \subseteq (A_f\ op\ B_f)$. Ce dernier intervalle n'est pas nécessairement représentable sur machine, nous le représenterons par le plus petit intervalle flottant $(A_f\ op\ B_f)_f$ qui contient cet intervalle réel. Ces deux passages à l'arithmétique flottante des intervalles nous mènent vers le principe de l'arithmétique flottante des intervalles [**?**] :

$$\alpha \in A, \beta \in B \Rightarrow \alpha\ op\ \beta \in (A_f\ op\ B_f)_f \tag{7.11}$$

Le principe de base de l'arithmétique réelle des intervalles et aussi l'arithmétique flottante des intervalles est le fait que le résultat soit toujours encadré par un intervalle. Cette propriété permet, dans les algorithmes de résolution d'équations, d'avoir des solutions *encadrées par intervalles* et non pas des solutions approchées.

En conclusion, en analyse par intervalles, un problème est résolu en trois étapes [**?**] :

1. la théorie est faite sur l'arithmétique des intervalles ;
2. le calcul est fait sur l'arithmétique flottante des intervalles ;
3. le principe d'inclusion (7.11) garantit la validité de la transition de l'arithmétique réelle des intervalles à l'arithmétique flottante des intervalles.

Plusieurs bibliothèques de l'arithmétique flottante des intervalles ont vu le jour, par exemple [**?**, **?**].

7.6.1 Extension des fonctions sur les intervalles

Soit $f : I\!R^n \rightarrow I\!R^m$.

6. En pratique, l'encadrement est implémenté en exploitant les deux fonctions *roundnear* et *rounddown*.

$exact(f)(X)$, avec $X \in I\!I(I\!R)^n$, est l'intervalle exact de variation de la fonction f dans l'intervalle X :

$$exact(f)(X) = \square\{f(x)|x \in X\}.$$

Une fonction $F : I\!I(I\!R)^n \to I\!I(I\!R)^m$ est appelée fonction d'inclusion de f si et seulement si :

$$\{f(x)|x \in X\} \subseteq F(X) \text{ pour tout } X \in I\!I(I\!R)^n \tag{7.12}$$

Les fonctions d'inclusion pour le cas matriciel sont définies de la même façon.

Le calcul de $exact(f)(X)$ est précis s'il n'existe pas d'occurrence multiple de variable (voir la section 7.6.2). Mais dans le cas général, le calcul de $exact(f)(X)$ de $f(x), x \in X$ est un problème indécidable (voir [?]). Calculer l'intervalle $\tilde{X}$, une approximation de $exact(f)(X) = Xe, X \in I\!I(I\!R)$ à ϵ près, défini par :

$$|\underline{\tilde{X}} - \underline{Xe}| \leq \epsilon \text{ et } |\overline{\tilde{X}} - \overline{Xe}| \leq \epsilon,$$

est un problème NP-difficile (voir [?]). En conséquence, les différentes fonctions d'inclusion que nous allons proposer sont des approximations *larges* de $exact(f)(X)$.

La définition 15 nous donne un moyen pour caractériser le comportement asymptotique des fonctions d'inclusion.

Définition 15 (ordre de convergence d'une extension)**.** *[?]*

Soit $F(X)$ l'extension sur les intervalles de $f : I\!R^n \to I\!R^m$ évaluée sur un domaine X. S'il existe une constante K, indépendante du domaine $X \in I\!I(I\!R)$ tel que

$$wid(F(X)) - wid(exact(f)(X)) \leq K wid(X)^\alpha$$

pour tous les domaines X avec $wid(X)$ suffisamment petit et un $\alpha > 0$ fixe, nous dirons que F est une fonction d'inclusion d'ordre α. Si α est égal à 1 (resp. à 2), alors nous dirons que l'inclusion est linéaire ou de premier ordre (resp. quadratique ou de deuxième ordre).

Exemple 4. *Voir les exemples donnés dans la sous-section* **??** *sur l'ordre de convergence d'une suite de nombres dans le cas général.*

7.6.2 Extension naturelle des fonctions

Soit $f : I\!R^n \to I\!R$ une fonction dont le calcul est défini sous forme d'une expression $f(x), x \in I\!R^n$, qui est composée des éléments suivants :

1. La variable x (ou une de ses composantes $\{x_1, \ldots, x_n\}$).
2. Des constantes réelles.
3. Les quatre opérations $\{+, -, \times, /\}$ de base de $I\!R$.
4. Des fonctions prédéclarées, $\{g_i\}_{i=1}^q$.
5. Des symboles syntaxiques (parenthèses, etc.).

L'extension naturelle sur les intervalles de la fonction f sur $X \in I\!I(I\!R)$ est définie [?] comme l'expression obtenue a partir de $f(x)$ en remplaçant :

1. chaque occurrence du vecteur réel x par le vecteur d'intervalles X,
2. chaque composante x_i par la composante intervalle X_i,
3. les opérations arithmétiques sur $\mathbb{R}$ par leurs correspondants dans l'arithmétique des intervalles,
4. et chaque occurrence d'une fonction prédéclarée g_i par son extension sur les intervalles G_i.

L'extension naturelle de $f(x)$, avec $x = (x_1, \ldots, x_n)$, de variables réelles $\{x_1, \ldots, x_n\}$ sur les intervalles est dénotée par $nat(f)(X)$, avec $X = (X_1, \ldots, X_n)$, de variables intervalles $\{X_1,\ldots,X_n\}$. Par construction, nous avons la propriété :

Théorème 8 (théorème fondamental de l'arithmétique des intervalles)**.**
[?] Soit $nat(f)(X)$ l'extension naturelle sur les intervalles de $f(x)$. Alors $nat(f)(X)$ contient toutes les valeurs de $f(x)$ pour tout $x \in X$.

Le théorème 8 est qualifié par [**?**, **?**] de théorème fondamental de l'arithmétique des intervalles.

Il existe plusieurs formes syntaxiques possibles de l'extension naturelle d'une fonction, qui donnent lieu à différents résultats.

Exemple 5. *[?]*
Soit l'expression :

$$e(x) = \frac{x+1}{x}, \ \textit{avec } x \in \mathbb{R}$$

nous pouvons extraire les deux extensions naturelles :

$$e_1(X) = 1 + \tfrac{1}{X}, \ \textit{avec } X \in \mathbb{I}(\mathbb{R})$$
$$e_2(X) = \tfrac{X+1}{X}, \ \textit{avec } X \in \mathbb{I}(\mathbb{R})$$

L'évaluation des deux extensions naturelles sur l'intervalle $[1,2]$ donne :

$$e_1([1,2]) = 1 + \tfrac{1}{[1,2]} = [1.5, 2]$$
$$e_2([1,2]) = \tfrac{[1,2]+1}{[1,2]} = [1, 3]$$

La valeur donnée par $e_1([1,2])$ est plus précise que celle de $e_2([1,2])$, car e_2 contient une occurrence multiple de la variable X.

Quand il n'y a pas d'occurrence multiple de variables, l'évaluation donne des intervalles précis, plus généralement nous avons :

Théorème 9 (expression sans occurrence multiple de variables)**.**
Soit $f : \mathbb{R}^n \to \mathbb{R}$ dont l'expression est une suite d'opérations utilisant seulement les quatre opérateurs standards. Supposons que dans cette expression chaque variable a une seule occurrence. Alors, l'évaluation de l'extension naturelle $nat(f)(X)$ de l'expression de f dans un domaine X est égale à $exact(f)(X)$.

Lors de l'établissement d'une extension naturelle, il est souhaitable d'éviter des occurrences multiples de variables.

Dans le cas général, l'établissement de l'extension naturelle optimale est un problème ouvert dans l'analyse par intervalles. Il est donc difficile de choisir, parmi un ensemble d'extensions naturelles, l'extension qui donnera le résultat le plus précis. Le choix est fait généralement à partir de considérations heuristiques.

Le choix de l'extension naturelle a des conséquences décisives sur les performances du calcul scientifique utilisant l'arithmétique des intervalles. Ainsi, il est important de disposer d'un prétraitement pour essayer de transformer l'extension naturelle en une autre extension naturelle plus précise.

Théorème 10 (convergence linéaire de l'extension naturelle)**.**
[?]
L'extension naturelle d'une fonction sur les intervalles est d'ordre 1.

Le théorème 10 montre que la surestimation de l'extension naturelle $nat(f)(X)$ est bornée par $wid(X)$.

7.6.3 Extension de Taylor

Soit $f : \mathcal{D}' \subseteq I\!R^n \to I\!R$ une fonction dérivable à dérivées continues ou de classe C1. Soient $\mathcal{D} \subseteq \mathcal{D}'$ et le point $c \in \mathcal{D}$. L'extension de la valeur intermédiaire de f sur le vecteur d'intervalles $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ en un point $X \in I\!I(I\!R)^n$ et centrée sur c est définie par

$$tay(f)(\mathcal{D}, X) = \begin{cases} f(c) + F'(\mathcal{D})(X - c) \\ \textbf{ou} \\ f(c_1, \ldots, c_n) + \sum_{i=1}^{n} \frac{\partial F}{\partial X_i}(\mathcal{D})(X - c)_i \end{cases} \tag{7.13}$$

avec $F'(\mathcal{D})$ l'évaluation du gradient de f sur le vecteur d'intervalles $\mathcal{D}$.

Le gradient $F'(\mathcal{D})$ est calculé en utilisant une extension (naturelle par exemple) sur les intervalles de la dérivée scalaire. Soit $x \in X$, la formule (7.13) est aussi l'extension sur les intervalles de la formule de Taylor d'ordre 1 :

$$f(c) + (x - c)^T f'(\xi) \in f(c) + F'(\mathcal{D})(X - c)$$

où ξ est un point entre x et c.

La propriété fondamentale de cette extension est issue du théorème 11.

Théorème 11 (convergence quadratique de l'extension de Taylor)**.**
[?]
Supposons que les composantes de F' sont des extensions de f' d'ordre un. Alors $tay(F)(\mathcal{D}, X)$ est une extension d'ordre 2 de f.

Le théorème 11 montre que la surestimation de l'extension de Taylor $tay(F)(\mathcal{D}, X)$ est bornée par $wid(X)^2$. La formule des valeurs intermédiaires est donc efficace sur des domaines de petite taille. Sur des domaines de grande taille — plus précisément de largeur supérieure à un —, la formule des valeurs intermédiaires croit plus rapidement vers ∞ que l'extension naturelle sur les intervalles quand $wid(X) \to \infty$. En conséquence, il est préférable d'utiliser l'extension naturelle sur les domaines de taille importante, et la formule de Taylor sur les domaines de taille réduite. [**?**] proposent

l'heuristique suivante : utiliser la formule de Taylor d'une fonction sur un intervalle X seulement quand $wid(X) \leq 1/2$.

Nous pouvons définir une extension généralisée de la formule de Taylor d'ordre k par :

Définition 16 (extension de Taylor d'ordre k)**.** *La formule (fonction) de Taylor d'ordre $k \geq 1$ dans un domaine $\mathcal{D}$ de la fonction réelle $f(x)$, à n variables x_i, avec $i = 1 \ldots n$, est définie par*

$$tay^k(f)(X) = f(c) + \sum_{\lambda=1}^{k-1} \sum_{i_1,\ldots,i_\lambda} \frac{1}{\lambda!} \frac{\partial^\lambda f}{\partial x_{i_1} \ldots \partial x_{i_\lambda}}(c)(X-c)_{i_1} \ldots (X-c)_{i_\lambda} + \sum_{i_1,\ldots,i_k} \frac{1}{k!} \frac{\partial^k F}{\partial X_{i_1} \ldots \partial X_{i_k}}(\mathcal{D})(X-c)_{i_1} \ldots (X-c)_{i_k}$$

avec $X \in I\!I(\mathcal{D})$, $c = mid(X)$ et $x \in X$.

L'instance d'ordre 1, $tay^1(f)(X)$, est la formule des valeurs intermédiaires (7.13). L'extension de Taylor d'ordre $k > 1$ est aussi de convergence quadratique (voir [?]).

En pratique, à partir de l'ordre 2, l'extension sur les intervalles de la formule de Taylor devient trop coûteuse.

7.7 Analyse par intervalles

Cette section est une introduction aux méthodes d'analyse par intervalles (voir [?, ?, ?, ?] pour plus de détails). On limitera notre présentation aux méthodes basées sur Newton par intervalles pour résoudre des systèmes multi-variés d'équations nonlinéaires.

L'objectif est de trouver les zéros d'un système de n équations $f_i(x_1, \ldots, x_n)$ avec n inconnues x_i dans un vecteur d'intervalles $X = \{X_1, \ldots, X_n\}$ où $x_i \in X_i$ pour $i = 1, \ldots, n$.

En premier, soit le cas des équations linéaires définies comme suit :

$$\mathbf{A}.x = \mathbf{b} \tag{7.14}$$

avec $\mathbf{A}$ une matrice d'intervalles et $\mathbf{b}$ un vecteur d'intervalles. Résoudre ce système linéaire sur les intervalles nécessite la détermination d'un vecteur d'intervalles X_0 contenant toutes les solutions pour tous les systèmes linéaires classiques (scalaires) dénotés $A.x = b$ où $A \in \mathbf{A}$ et $b \in \mathbf{b}$. Trouver X_0 est un problème difficile, deux méthodes basiques par intervalles fournissent une surestimation du vecteur par intervalles X_1 contenant X_0.

Notons que l'algorithme d'élimination de Gauss fonctionne sur les systèmes linéaires sur les intervalles. De ce fait, il est possible d'obtenir X_1 si les pivots de Gauss ne contiennent pas zéro dans le processus de triangulation. C'est pourquoi, en général, l'interval calculé est assez large et l'étape de préconditionnement est nécessaire. En d'autres termes, on doit multiplier les deux cotés de l'équation 7.14 avec l'inverse de la matrice milieu de $\mathbf{A}$. La matrice $m(\mathbf{A})^{-1}\mathbf{A}$ est alors "très proche" de la matrice identité et la largeur de X_1 est assez petite [?].

Une autre méthodes bien connue dans la résolution des systèmes linéaires sur les intervalles, est la méthode itérative de Gauss-Seidel. Pour toute inconnue X_i, l'algorithme [?] est défini avec le processus itératif suivant :

$$X_i^{k+1} = (\mathbf{b}_i - \sum_{j=1}^{i-1} \mathbf{A}_{i,j} X_j^{k+1} - \sum_{j=i+1}^{n} \mathbf{A}_{i,j} X_j^k)/\mathbf{A}_{i,i} \cap X_i^k \tag{7.15}$$

Ici aussi, une étape de préconditionnement permettrait de réduire la taille des intervalles. Les détails sur l'implémentation de cet algorithme sont explicités dans [?]. Remarquons que cette méthode est très proche des méthodes de filtrage proposées en programmation par contraintes (voir le chapitre sur l'optimization globale).

Une autre alternative consiste à utiliser la méthode de Krawczyk. Cette méthode nécessite aussi un préconditonnement du système (see [?, ?]).

Pour résoudre un système nonlinéaire, l'algorithme de Newton est le plus souvent utilisé. Ci-dessous le schéma général :

$$X_{k+1} = N(\tilde{x}_k, X_k) \cap X_k \quad \text{with} \quad N(\tilde{x}_k, X_k) = \tilde{x}_k - A.f(\tilde{x}_k)) \tag{7.16}$$

A est une matrice d'intervalles qui contient toutes les inverses de la matrice Jacobienne du système F. Le scalaire $\tilde{x}_k$ doit être choisie dans X_k (par exemple le milieu X_k). Ainsi, les propriétés suivantes[7] sont vérifiées :

— Si $N(\tilde{x}_k, X_k) \cap X_k = \emptyset$, alors le système F a une seule solution dans X_k.

— Si $N(\tilde{x}_k, X_k)_k \cap X_k \subset X_k$, il existe une seule ou plusieurs solutions dans X_{k+1}.

La détermination de la matrice A est une étape délicate dans les algorithmes proposés. La matrice A est l'inverse de la matrice jacobienne $A = [F^{'}(X_k)]^{-1}$ évaluée sur le vecteur des intervalles X_k. C'est pourquoi, inverser la matrice sur les intervalles peut être couteuse en temps et en espace. L'alternative consiste à résoudre le système linéaire $F^{'}(X_k)(N(\tilde{x}_k, X_k) - \tilde{x}_k) = -f(\tilde{x}_k)$ pour déterminer $N(\tilde{x}_k, X_k)$. Ce travail peut être entretenu par l'un des algorithmes cités précédemment (see [?]).

Le schéma de Krawczyk offre des alternatives intéressantes. Il est défini par le schéma itératif suivant :

$$X_{k+1} = K(\tilde{x}_k, X_k) \cap X_k \text{ with } K(\tilde{x}_k, X_k) = \tilde{x}_k - [f(\tilde{x}_k)]^{-1} f(\tilde{x}_k) + (I - [f(\tilde{x}_k)]^{-1} F^{'}(X_k))(X_k - \tilde{x}_k) \tag{7.17}$$

Les propriétés de ce schéma sont utilisées par Moore [?] pour vérifier l'existance et l'unicité du zéro et la convergence de ce schéma. Notons que ce schéma utilise l'inverse de la matrice scalaire. Cette méthode est particulièrement rapide et efficace sur des intervalles réduits.

7. Pour les autres propriétés et l'implementation, voir [?]

Bibliographie

[Aribi, 2014] Aribi, N. (2014). Contribution à l'élicitation des paramètres en optimisation multicritère. Technical report, Thèse Docteur en Sciences, Université Oran1, Université de Nice-Sophia Antipolis.

[Bastin, 2010] Bastin, F. (2010). Modèles de recherche opérationnelle. Technical report, Support de cours, Département d'Informatique et de Recherche Opérationnelle https://www.iro.umontreal.ca/~bastin/Cours/IFT1575/IFT1575.pdf.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In Harrison, M. A., Banerji, R. B., and Ullman, J. D., editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM.

[Cori et al., 2001] Cori, R., Hanrot, G., and Steyaert, J.-M. (2001). Conception et analyse des algorithmes. Technical report, Ecole polytechnique.

[Cormen et al., 1994] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1994). *Introduction à l'algorithmique (traduit par Xavier Cazin)*. Dunod.

[Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms, Second Edition*. The MIT Press.

[Gaudel et al., 1990] Gaudel, M.-C., Froidevaux, C., and Soria, M. (1990). *Types de données et algorithmes*. McGraw-Hill, Paris.

[Levin, 1973] Levin, L. A. (1973). Universal search problems (??????????????????????????). *Problems of Information Transmission (??????????????????????????)*, 9(3).

[Papadimitriou et al., 2006] Papadimitriou, C. H., Dasgupta, S., and Vazirani, U. (2006). *Algorithms*. McGraw Hill.

[Prins, 1994] Prins, C. (1994). *Algorithmes de graphes*. Eyrolles.

[R. Faure, 1995] R. Faure, B. Lemaire, C. P. (1995). *Précis de recherche opérationnelle : Méthodes et exercices d'application*. Eyrolles.