

TP de méthodes de résolution de problèmes NP-complets

Table des matières

1	Partie théorique	2
1.1	Sur le problème de la couverture sommet minimale	2
1.2	Sur le problème de la couverture d'ensembles	6
1.3	Sur le problème du couplage maximum de poids minimum	7
1.4	Sur le problème de la coupe maximum	9
1.5	Sur le problème de partition	10
1.6	Sur le problème du sac à dos simple	12
1.7	Programmation dynamique	13
1.8	Sur le produit matriciel	16
1.9	Résolution numérique par la méthode primal-dual	19
1.10	Seuil d'approximation pour le problème Bin Packing	19
1.11	Seuil d'approximation pour le problème de la coloration de sommets (resp. d'arêtes)	21
1.12	Comparaison de <i>branch and bound</i> et <i>branch and cut</i>	22
2	Partie pratique	35
2.1	Programmation dynamique	35
2.2	Branch and bound	38
2.3	Comparaison du Branch&Bound et de la programmation dynamique sur l'exemple du TSP	38

Chapitre 1

Partie théorique

1.1 Sur le problème de la couverture sommet minimale

1.1.1 Première approche : la programmation linéaire en nombres entiers

(a) Justification de l'utilisation de la Programmation Linéaire en Nombres Entiers

On considère le problème de la couverture minimale sous la forme suivante :

$$\left\{ \begin{array}{l} \min z = \sum_{j=1}^n x_j \\ x_r + x_s \geq 1, \quad \forall \{v_r, v_s\} \in E \\ x_j \in \{0, 1\} \quad j = 1, \dots, n \end{array} \right.$$

La fonction objectif représente le nombre de sommets utilisés par la solution du problème. Le fait de minimiser la fonction objectif permet d'assurer la couverture minimale. Chaque clause est relative à une arête du graphe, et impose qu'au moins un des sommets adjacents à cette arête soit dans la couverture.

On a donc bien un problème de Programmation Linéaire en Nombres Entiers permettant de résoudre le problème de la couverture minimale.

(b) Justification des clauses

Considérons le graphe donné par la figure 1.1.1.

Sur ce graphe, le programme linéaire en nombres entiers est le suivant :

$$\left\{ \begin{array}{l} \min z = x_A + x_B + x_C \\ x_A + x_B \geq 1 \\ x_A + x_C \geq 1 \\ x_B + x_C \geq 1 \\ x_A, x_B, x_C \in \{0, 1\} \end{array} \right.$$

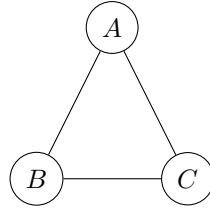


FIGURE 1.1 – Exemple

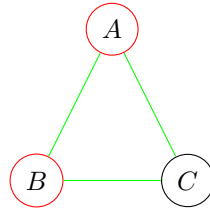


FIGURE 1.2 – Solution

Il est très simple ici de comprendre pourquoi il est impossible de considérer le programme linéaire suivant :

$$\left\{ \begin{array}{l} \min z = x_A + x_B + x_C \\ x_A + x_B = 1 \\ x_A + x_C = 1 \\ x_B + x_C = 1 \\ x_A, x_B, x_C \in \{0, 1\} \end{array} \right.$$

Ce programme ne permet pas de résoudre la couverture minimale sur le graphe donné par la figure 1.1.1. Quelque soit le sommet choisi dans un premier lieu pour appartenir à la couverture minimale, il est impossible d'en choisir un second pour compléter cette dernière. Prenons un exemple, nous forçons le sommet A à appartenir à la couverture minimale (respectivement B et C). Ce choix force : $x_B = 0$ et $x_C = 0$ (respectivement, $x_A = 0$ et $x_C = 0$, et $x_A = 0$ et $x_B = 0$). Il est donc impossible de respecter la clause $x_B + x_C = 1$, le problème (au vu de sa modélisation) n'aurait donc pas de solution, or le graphe de la figure 1.1.1 montre le contraire.

(c) Une borne inférieure des solutions optimales

On cherche à montrer qu'une solution optimale du programme linéaire en nombres entiers est une borne inférieure de toute solution optimale du programme relaxé. Raisonnons par l'absurde et considérons une solution optimale du programme linéaire, notée n^* telle qu'il existe x^* solution optimale du problème relaxé vérifiant $x^* < n^*$. Toute solution du programme linéaire est solution du programme relaxé¹. Ceci implique : n^* solution du programme relaxé, et donc $x^* < n^*$ impossible. On a donc : $n^* \leq x^*$ ce qui est la définition d'une borne inférieure.

1. Une solution appartenant à \mathbb{N} appartient aussi à \mathbb{R}

(d) A propos de la relaxation de contrainte

Pour démontrer que la relaxation des contraintes d'intégrité implique $x_r \geq \frac{1}{2}$ ou $x_s \geq \frac{1}{2}$, le raisonnement par l'absurde sera utilisé. Soient x_s et x_r les variables relatives aux sommets r et s adjacents à l'arête (rc) et telles que, après relaxation des contraintes, on a : $x_r < \frac{1}{2}$ et $x_s < \frac{1}{2}$. On en déduit donc que $x_r + x_s < 1$ et donc la contrainte liée à l'arête (rs) est violée, l'hypothèse de départ est donc fausse. On a donc, $\forall (rs) \in V : x_r \geq \frac{1}{2}$ et $x_s \geq \frac{1}{2}$.

(e) Une 2-approximation

Mettons en évidence le pire des cas pouvant se présenter : pour une arête $(rs) \in V$, un seul sommet est nécessaire pour la couverture de cette dernière dans le cas de la couverture minimale, mais l'algorithme approché retourne : $x_r = x_s = \frac{1}{2}$. Après la phase d'arrondis, on a $x_r = x_s = 1$ et donc les deux sommets appartiennent à la solution approchée, cette phase multiplie donc au pire le nombre de sommets (pour chaque clause par 2), ce qui implique que le cardinal de la solution approximée est au plus 2 fois la solution optimale.

Cet algorithme est donc une 2-approximation.

(f) Dans le cas d'un graphe valué

i. Programme linéaire

$$\begin{cases} \min z = \sum_{(i,j) \in E} (x_i \times w_{ij} + x_j \times w_{ij}) \\ x_r + x_s \geq 1, \quad \forall \{v_r, v_s\} \in E \\ x_j \in \{0, 1\} \quad j \in \{1, \dots, n\} \end{cases}$$

ii. PLNE avec matrice

Posons la matrice A , la (n, n) -matrice d'adjacente du graphe et X le vecteur contenant les variables existentielles définies précédemment. Le PLNE correspondant est le suivant :

$$\begin{cases} \min z = \sum_{(i,j) \in E} (x_i \times w_{ij} + x_j \times w_{ij}) \\ A.X \geq 1 \\ x_j \in \{0, 1\} \quad j \in \{1, \dots, n\} \end{cases}$$

iii. Algorithme 2-approché Dans le pire des cas, on prend tous les sommets du graphe et chaque arête est comptée deux fois. La solution optimale admet une borne inférieure égale à la somme du poids des arcs. Donc la valeur de la fonction objectif dans le pire des cas vaut deux fois la somme totale des poids des arêtes et il existe un algorithme deux-approché (celui qui prend tous les sommets).

1.1.2 Seconde approche : la recherche d'un couplage maximal

(a) Une 2-approximation

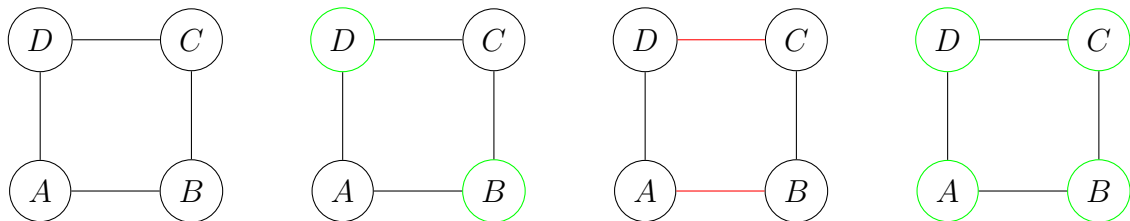
Commençons par prouver que l'algorithme retourne une couverture des arêtes par les sommets. Considérons donc une arête (rs) non couverte par l'ensemble de sommets retourné par l'algorithme, par définition du couplage, il serait donc possible d'ajouter (rs) au couplage. Or le couplage calculé par l'algorithme est maximal, on en déduit que l'arête (rs) telle qu'elle est définie ne peut exister et donc que l'ensemble de sommets obtenu couvre l'ensemble des arêtes du graphe.

Appelons c le couplage calculé par l'algorithme et x^* la solution optimale du problème de la couverture par les sommets, on sait que $\text{Card}c \leq \text{Card}x^*$, or pour construire la solution approchée, on ajoute à C les deux extrémités des arêtes utilisées pour le couplage. On a donc :

$$\begin{aligned} \text{Card}C &= 2 \times \text{Card}c \leq 2 \times \text{Card}x^* \\ \Rightarrow \frac{\text{Card}C}{\text{Card}x^*} &= 2 \end{aligned}$$

(b) Exemple de graphe pathologique

Le graphe suivant met en évidence la borne 2 de l'algorithme.



Le graphe Une couverture minimale Un couplage maximale La solution renvoyée

(c) Application de l'algorithme

L'application de l'algorithme 3 au graphe d'exemple de la figure 1 donne une couverture $C = \{a_i\} \cup \{b_i\}$ alors que la solution optimale est $C = \{b_i\}$. Si le graphe de la figure 1 est bien le pire des cas pour cet algorithme alors c'est un algorithme 1.6-approché.

1.2 Sur le problème de la couverture d'ensembles

1.2.1 Modélisation du problème à l'aide de la PLNE

On considère le problème de la couverture d'ensemble, défini par : Soit $E = \{e_1, \dots, e_n\}$, soient S_1, \dots, S_m des sous-ensembles non vides de E tels que $\forall i \in \{1, \dots, m\}$, on a : $S_i \subset E$. On associe à chaque ensemble S_j un poids $w_j \geq 0$. Le problème consiste à trouver une collection de sous-ensemble de poids minimum et telle que $\bigcup_{i=1}^m S_i = E$.

Ce problème peut s'exprimer à l'aide de la Programmation Linéaire en Nombres entiers de la manière suivante :

$$\begin{cases} \min z = \sum_{i=1}^m w_i x_i \\ \sum_{x_j: e_i \in S_j} x_j \geq 1 \quad \forall e_i \in E \\ x_i \in \{0, 1\} \end{cases}$$

1.2.2 Procédure d'arrondis

Soit $f = \max_{i=1, \dots, n} f_i$, avec f_i le cardinal de l'ensemble des sous-ensembles de E contenant e_i : $f_i = |\{j : e \in S_j\}|$.

Définissons la procédure d'arrondis suivante :

$$x_i = \begin{cases} 1 & \text{si } x_i \geq \frac{1}{f} \\ 0 & \text{sinon} \end{cases}$$

1.2.3 Garantie d'une solution réalisable

Cherchons à démontrer que cette procédure d'arrondis garantit une solution réalisable. Pour ce faire, nous allons procéder par l'absurde. Supposons qu'il existe un sommet k qui ne respecte pas sa contrainte d'intégrité associée, à savoir :

$$\sum_{x_j: e_k \in S_j} < 1$$

Les variables x_j de cette contrainte étant définies positives et entières, le cas pris en considération si dessus implique que toutes les variables de l'inéquation sont nulles pour le programme en nombres entiers et donc :

$$\forall x_j : e_k \in S_j < \frac{1}{f}$$

dans le cas de la version relaxée du problème. Or ceci n'est possible que si le nombre de sous-ensemble contenant x_k est supérieur à f , ce qui est impossible par définition. Tous les sommets respectent donc leur contrainte d'intégrité et on en déduit que la procédure d'arrondis garantit une solution réalisable.

1.2.4 Existence d'un algorithme f -approché

Considérons l'algorithme 1

Algorithme 1 Approximation couverture par ensemble

- 1: Exprimer le problème en programmation linéaire en nombres entiers
 - 2: Résoudre la version relaxée
 - 3: Réaliser la procédure d'arrondis
-

En utilisant la procédure d'arrondis étudiée plus haut, on sait que pour chaque clause, il existe x_k tel que $x_k \geq \frac{1}{f}$, au pire des cas², chaque x_k est multiplié par f , ce qui implique que la solution obtenue est au plus f fois plus grande que la solution optimale. Il s'agit donc d'un algorithme f -approché.

1.2.5 Cas ou $f = 2$

Si $f = 2$ alors le problème devient une couverture minimum par sommets.

1.3 Sur le problème du couplage maximum de poids minimum

1.3.1 Modélisation du problème

Soit un graphe $G = (V, E)$ avec V l'ensemble de ses sommets et E l'ensemble de ses arêtes. Soit $\{\forall(i, j) \in E, X_{(i,j)}\}$ un ensemble de variables booléennes qui indiquent le choix de l'arête (i, j) correspondante dans le couplage. Soit $P_{(i,j)}$ le poids de l'arête (i, j) .

La première modélisation intuitive est la suivante :

Maximiser

$$\sum_{(i,j) \in E} (X_{(i,j)} \times H) - \sum_{(i,j) \in E} (X_{(i,j)} \times P_{(i,j)})$$

Sous contraintes

$$\forall i \in V, \sum_{(i,j) \in E} X_{(i,j)} + \sum_{(j,i) \in E} X_{(j,i)} \leq 1$$

$$\forall(i, j) \in E, X_{(i,j)} \in \{0, 1\}$$

$$\forall(i, j) \in E, P_{(i,j)} \geq 0$$

avec H une constante très grande. Cependant, après quelques instants de réflexion, nous pouvons imaginer une modélisation plus élégante :

Minimiser

$$\sum_{(i,j) \in E} (X_{(i,j)} \times P_{(i,j)})$$

2. cas similaire à celui de l'exercice 1

Sous contraintes

$$\begin{aligned} \forall i \in V, \sum_{(i,j) \in E} X_{(i,j)} + \sum_{(j,i) \in E} X_{(j,i)} &= 1 \\ \forall (i,j) \in E, X_{(i,j)} &\in \{0,1\} \\ \forall (i,j) \in E, P_{(i,j)} &\geq 0 \end{aligned}$$

En effet, les contraintes forcent le couplage à être maximum tandis que la fonction objectif le force à tendre vers le poids minimum.

1.3.2 Modélisation du problème appliquée au graphe de la figure 2

Minimiser

$$\epsilon \times X_{ab} + \epsilon \times X_{bc} + \epsilon \times X_{ac} + M \times X_{ae} + M \times X_{cd} + M \times X_{bf} + \epsilon \times X_{df} + \epsilon \times X_{de} + \epsilon \times X_{fe}$$

Sous contraintes

$$\begin{aligned} X_{ab} + X_{ac} + X_{ae} &= 1 \\ X_{ab} + X_{bc} + X_{bf} &= 1 \\ X_{ac} + X_{bc} + X_{cd} &= 1 \\ X_{cd} + X_{de} + X_{df} &= 1 \\ X_{ae} + X_{de} + X_{df} &= 1 \\ X_{ef} + X_{df} + X_{bf} &= 1 \\ \forall (i,j) \in E, X_{(i,j)} &\in \{0,1\} \\ \epsilon &\geq 0 \\ M &\geq 0 \end{aligned}$$

1.3.3 Solution optimale entière $z(ILP)$

Sur un exemple de cette taille, il est facile de trouver une solution à la main. Il y a plusieurs solutions optimales de poids total $M + 2\epsilon$ sur cet exemple ; l'une d'entre elles est le couplage $\{(a,b), (c,d), (e,f)\}$.

La résolution de ce PLNE par *glpsol* (solveur de *GLPK*) donne bien la même solution.

1.3.4 Solution optimale $z(LP)$ pour le programme relaxé

Relaxer le programme revient à transformer la contrainte d'intégrité des $X_{(i,j)}$ en la contrainte suivante :

$$\forall (i,j) \in E, 0 \leq X_{(i,j)} \leq 1$$

En rentrant le PL relaxé dans *glpsol*, nous obtenons le couplage de poids total 3ϵ :

$$\{X_{ab} = 0.5; X_{ac} = 0.5; X_{bc} = 0.5; X_{df} = 0.5; X_{ef} = 0.5; X_{de} = 0.5\}$$

1.3.5 Conclusion sur la pertinence de la formulation

La solution trouvée au PLNE étant optimale, la formulation du problème semble être pertinente.

1.4 Sur le problème de la coupe maximum

1.4.1 Complexité

A chaque itération de l'algorithme, la valeur de la coupe maximale augmente au minimum d'une unité. Or la valeur de la coupe maximale étant bornée par le nombre d'arêtes, on obtient donc que l'algorithme effectue au plus m opérations. On a donc un algorithme en $O(m)$.

1.4.2 Un algorithme 2-approché

Considérons (Y_1, Y_2) la coupe renvoyée par l'algorithme, nous chercherons dans un premier temps à montrer que chaque sommet dans Y_1 admet au moins autant d'arêtes dans Y_2 que dans Y_1 .

Pour ce faire, considérons un sommet $v \in Y_1$, supposons que ce sommet possède plus d'arêtes dans Y_1 que dans Y_2 , nous noterons a_1 le nombre d'arêtes incidentes à v dans Y_1 et a_2 le nombre d'arêtes adjacentes à v dans Y_2 . Déplacer v de Y_1 vers Y_2 reviendrait à diminuer la coupe de a_2 et augmenter celle-ci de a_1 , or d'après l'hypothèse de départ $a_2 < a_1$, on observerait une augmentation de la valeur de la coupe, ce qui est impossible si (Y_1, Y_2) est une coupe retournée par l'algorithme. On en déduit donc que l'hypothèse de départ est fausse.

Autrement dit, pour un sommet $v \in V$, en notant d_v le degré de v , v_c le nombre d'arêtes adjacentes à v traversant la coupe et v_s le nombre d'arêtes adjacentes à v ne la traversant pas, on peut écrire :

$$\begin{aligned} v_c + v_s &= d_v \quad \text{or on a } v_c \geq v_s \\ \Rightarrow \quad v_c &\geq \frac{d_v}{2} \end{aligned}$$

Si l'on généralise sur l'ensemble du graphe, on peut en déduire :

$$\begin{aligned} |(Y_1, Y_2)| &= \frac{1}{2} \sum_{v \in V} v_c \\ &\geq \frac{1}{2} \sum_{v \in V} \frac{d_v}{2} \\ &= \frac{m}{2} \end{aligned}$$

On a donc : $|(Y_1, Y_2)| \geq \frac{m}{2}$, de plus comme vu précédemment, la valeur de la coupe maximale (que nous noterons OPT) est bornée par le nombre d'arêtes. On peut donc en déduire :

$$\frac{OPT}{|(Y_1, Y_2)|} \leq \frac{m}{\frac{m}{2}} = 2$$

L'algorithme donné est donc bien un algorithme 2-approché.

1.5 Sur le problème de partition

Soit un ensemble d'objets $A = \{a_i : i \in [1; n]\}$ et une fonction de poids $p : A \rightarrow \mathbb{N}$.

Le problème de décision est le suivant :

existe-t-il $Y_1, Y_2 \subseteq A$ $Y_1 \cap Y_2 = \emptyset$: $\sum_{y_1 \in Y_1} p(y_1) = \sum_{y_2 \in Y_2} p(y_2) = P$?

Son problème d'optimisation associé est défini comme suit :

minimiser $\sum_{y_1 \in Y_1} p(y_1) : \sum_{y_1 \in Y_1} p(y_1) - \sum_{y_2 \in Y_2} p(y_2) \geq 0$.

On note :

- $w(E) = \sum_{e \in E} p(e)$ avec $E \subseteq A$,
- $L = \frac{w(A)}{2}$.

Soit l'algorithme suivant :

Algorithm 2 PTAS Partition

Require: $A = \{a_i : i \in [1, n]\}$ un ensemble de n objets, $p : A \rightarrow \mathbb{N}$ une fonction de poids sur ceux-ci, $r > 1$ le rapport d'approximation désiré

```

1 if  $r \geq 2$  then
2   return  $A, \emptyset$ 
3 end if
4 Trier  $A$  par ordre décroissant de  $p$ 
5 Soit  $(a_1, \dots, a_n)$ 
6  $k(r) \leftarrow \lceil \frac{2-r}{r-1} \rceil$ 
7 Première phase
8 Trouver une partition optimale  $Y_1, Y_2$  de  $a_1, \dots, a_{k(r)}$ 
9 for  $j = k(r) + 1$  à  $n$  do
10   if  $\sum_{y_1 \in Y_1} p(y_1) \leq \sum_{y_2 \in Y_2} p(y_2)$  then
11      $Y_1 \leftarrow Y_1 \cup \{a_j\}$ 
12   else
13      $Y_2 \leftarrow Y_2 \cup \{a_j\}$ 
14   end if
15 end for
16 return  $Y_1, Y_2$ 
```

1.5.1 r-approximation ; cas où $r \geq 2$

Montrons que $\forall r \geq 2$, la solution A, \emptyset est une r -approximation.

On sait que : $w(A) = w(Y_1) + w(Y_2)$ donc $w(Y_1) \geq \frac{w(A)}{2}$. Ie, $sol_{opt}(A) \geq \frac{w(A)}{2}$. La valeur de la solution A, \emptyset $sol_{algo}(A)$ vaut $w(Y_1) = w(A)$. On a alors $w(Y_1) = 2 \times \frac{w(A)}{2}$ et donc $w(Y_1) \leq 2 \times sol_{opt}(A)$.

Nous avons bien $sol_{algo}(A) \leq 2 \times sol_{opt}(A) \leq r \times sol_{opt}(A)$.

1.5.2 Cas où $r < 2$

Soit a_h le dernier élément inséré dans Y_1 .

(a)

Nous devons montrer que $w(Y_1) - L \leq \frac{p(a_h)}{2}$.

Posons $w(Y_1)'$ et $w(Y_2)'$ les poids respectifs deux partitions avant l'ajout de a_h .

Puisque a_h est ajouté à Y_1 , on sait que $w(Y_1)' \leq w(Y_2)'$. De plus, a_h est le dernier élément ajouté à Y_1 , donc $w(Y_1) = w(Y_1)' + p(a_h)$. Cela implique aussi que tous les éléments suivants sont ajoutés dans Y_2 , donc $w(Y_2)' \leq w(Y_2)$.

On a donc $w(Y_1)' = w(Y_1) - p(a_h) \leq w(Y_2)' \leq w(Y_2)$.

$w(Y_1) - w(Y_2) \leq p(a_h) \leftrightarrow w(Y_1) + w(A) - w(Y_2) - w(A) \leq p(a_h)$

$\leftrightarrow 2 \times w(Y_1) - w(A) \leq p(a_h) \leftrightarrow w(Y_1) - \frac{w(A)}{2} \leq \frac{p(a_h)}{2}$

$\leftrightarrow w(Y_1) - L \leq \frac{p(a_h)}{2}$.

(b)

Si a_h est inséré durant la première phase, l'algorithme renverra la solution optimale. En effet a_h est le dernier élément de Y_1 , donc tous les suivants seront insérés dans Y_2 , de plus $w(Y_1) \geq w(Y_2)$. Or, a_h est inséré dans la première phase, c'est à dire dans la phase de recherche optimale. On note Y_1', Y_2' les ensembles à la fin de celle-ci. On sait donc que $w(Y_1)'$ est la solution optimale du sous problème $Y_1' \cup Y_2'$. On a alors $w(Y_1) = w(Y_1)' = \text{sol}_{\text{opt}}(A')$ avec $A' = Y_1' \cup Y_2'$, puisque tous les éléments suivants sont insérés dans Y_2 , $\text{sol}_{\text{opt}}(A') \leq \text{sol}_{\text{opt}}(A)$.

On a donc $w(Y_1) \leq \text{sol}_{\text{opt}}(A)$ et $w(Y_1) \geq \text{sol}_{\text{opt}}(A)$ par définition : $\text{sol}_{\text{algo}}(A) = \text{sol}_{\text{opt}}(A)$.

(c)

Supposons maintenant que a_h est inséré durant la seconde phase, et intéressons nous aux $p(a_j) \forall j \in [1; k(r)]$. Puisque a_h est inséré dans la seconde phase ($h \geq k(r) + 1$), et que les a_i sont rangés par ordre décroissant de poids, on a : $p(a_h) \leq p(a_j) \forall j \in [1; k(r) + 1]$.

Ce qui implique : $\sum_{j=1}^{k(r)+1} p(a_h) \leq \sum_{j=1}^{k(r)+1} p(a_j)$.

Or, $\sum_{j=1}^{k(r)+1} p(a_j) \leq w(A)$.

On a donc $(k(r) + 1) \times p(a_h) \leq w(A) = 2 \times L$.

(d)

Toute solution optimale est minorée par $\frac{w(A)}{2}$.

(e)

Nous allons montrer que le ratio est bien majorée par r .
D'après la question 1.5.2, nous savons que $w(Y_1) - L \leq \frac{p(a_h)}{2}$. Donc $2 \times w(Y_1) - 2 \times L \leq p(a_h)$.
Or, par 1.5.2, nous avons : $p(a_h) \leq \frac{2 \times L}{k(r)+1}$.
On obtient alors $2 \times w(Y_1) - 2 \times L \leq 2 \times (r-1) \times L \leftrightarrow w(Y_1) \leq r \times L$.
On a bien : $sol_{algo}(A) \leq r \times sol_{opt}(A)$.

Nous pouvons en conclure que l'algorithme 2 renvoie bien une solution r -approchée.

1.5.3 Complexité

La complexité de l'algorithme est : $n \cdot \log(n) + 2^{k(r)+1} + n$

1.6 Sur le problème du sac à dos simple

1.6.1 Construction d'un algorithme approché

(a) Complexité de l'algorithme

La complexité de l'algorithme, une fois les nombres triés, est en $O(n)$. Cependant, il est possible de démontrer que la complexité des algorithmes de tris basés sur des fonctions de comparaisons ne peut être inférieures à $O(n \log n)$. la complexité de cet algorithme est donc similaire à la complexité de l'algorithme de tri utilisé.

(b) Minoration de $cost(T)$

Dans un premier temps nous montrerons que l'existence d'un indice j pour lequel $cost(T) + w_{j+1} > b$ n'est pas systématique, mais que si ce dernier n'existe pas, le problème n'existe pas non plus.

En effet, si cet indice n'existe pas, on a alors :

$$\sum_{i=1}^n w_i \leq b$$

On en déduit donc que la solution optimale est donnée par $T = \{1, \dots, n\}$. Ce qui en soit même n'a aucun intérêt³. Nous admettrons donc l'existence de cet indice j .

3. Le problème du sac à dos, dans cette configuration n'est plus NP-Complet

Supposons, à l'indice j , que $\text{cost}(T) \leq \frac{b}{2}$. Par définition de l'indice j , on a $\text{cost}(T) + w_{j+1} > b$, ce qui implique $w_{j+1} > \frac{b}{2}$ et donc :

$$\begin{aligned} w_{j+1} &> \text{cost}(T) \\ \Rightarrow w_{j+1} &> \sum_{i=1}^j w_i \\ \Rightarrow w_{j+1} &> w_i \end{aligned}$$

Or les éléments w_i étant triés, on a $w_j \geq w_{j+1}$. L'hypothèse de départ est donc fausse.

(c) Performance relative de l'algorithme

Notons OPT la solution optimale au problème du sac à dos, et A la solution approchée donnée par l'algorithme. Rappelons, par définition du problème, que $OPT \leq b$. La performance relative nous est donnée par :

$$\frac{OPT}{A} \leq \frac{b}{\frac{b}{2}} = 2$$

La performance relative de l'algorithme est donc 2.

1.6.2 Construction d'un schéma d'approximation

(a) Complexité de l'algorithme

De façon très grossière, la construction de l'ensemble des sous-ensembles S_k de k éléments se fait en $k \times n^k$ opérations. On a donc une construction des sous-ensembles réalisée en :

$$\sum_{i=1}^k i.n^i$$

Soit une construction en $O(n^k)$.

L'ensemble étant déjà trié, l'algorithme glouton s'exécute en $O(n)$, d'où une exécution totale de l'algorithme en $O(n^{k+1})$.

1.7 Programmation dynamique

1.7.1 Sur le problème de la partition

Condition nécessaire sur la somme des poids des n objets

Les poids de chacun des objets étant des valeurs entières, pour qu'il existe deux sous-ensembles distincts ayant le même poids, il faut que la somme des poids des objets soit paire.

Récurrance

Introduisons l'expression booléenne $T(i, j)$: *Etant donnés les i premiers éléments de la famille, il existe un sous-ensemble de ces i éléments de poids j .*

Dans un premier temps il paraît évident que si $j = 0$, le sous-ensemble existe : il s'agit de l'ensemble vide.

$$\forall i \in [0, n], \quad T(i, 0) = \text{VRAI} \quad (1.1)$$

De plus, si j est égal au poids de l'un des i premiers éléments, alors l'existence du sous-ensemble est avérée. Autrement dit, si l'on appelle K_i l'ensemble formé par les poids des objets i , on a :

$$\forall i \in [0, n], \quad T(i, j \in K_i) = \text{VRAI} \quad (1.2)$$

Troisièmement, l'ensemble des $i - 1$ premiers objets est un sous ensemble de l'ensemble des i premiers objets : $E_{i-1} \subset E_i$. On en déduit donc, que s'il existe un sous ensemble de E_{i-1} de poids j tel que $T(i - 1, j) = \text{VRAI}$, alors ce sous-ensemble est aussi un sous-ensemble de E_i tel que $T(i, j) = \text{VRAI}$ et ce par le principe d'inclusion.

$$\forall i \in [0, n], \quad \text{si } T(i - 1, j) = \text{VRAI}, \text{ alors } T(i, j) = \text{VRAI} \quad (1.3)$$

Enfin, considérons S_j un sous-ensemble de E_{i-1} de poids j vérifiant donc $T(i - 1, j) = \text{VRAI}$, alors $S_j \cup a_i$ est un sous-ensemble de E_i de poids $j + p(a_i)$ et donc vérifiant $T(i, j + p(a_i)) = \text{VRAI}$. On en déduit donc :

$$\forall i \in [0, n], \quad \text{si } T(i - 1, j - p(a_i)) = \text{VRAI}, \text{ alors } T(i, j) = \text{VRAI} \quad (1.4)$$

En réunissant les équations ci dessus dans une seule expression booléenne, on obtient la formule de la ligne i en fonction de la ligne $i - 1$ et $p(a_i)$:

$$T(i, j) = j == 0 \vee j == p(a_i) \vee T(i - 1, j) \vee T(i - 1, j - p(a_i)) \quad (1.5)$$

On peut alors donner un algorithme pour résoudre le problème de la partition : l'algorithme 3.

Algorithm 3 solve-partition

```

1:  $p_{max} \leftarrow \sum_{i \in [1, n]} p(a_i)$ 
2: if  $p_{max} \equiv 1 \pmod{2}$  then
3:   Pas de solutions
4: else
5:   for  $j \in [0, p_{max}/2]$  do
6:     if  $j = 0$  then
7:        $T(0, j) \leftarrow \text{true}$ 
8:     else
9:        $T(0, j) \leftarrow \text{false}$ 
10:    end if
11:  end for
12:  for  $i \in [1, n]$  do
13:    for  $j \in [0, p_{max}/2]$  do
14:       $T(i, j) \leftarrow [(j = 0) \vee (j = p(a_i)) \vee (T(i - 1, j)) \vee (T(i - 1, j - p(a_i)))]$ 
15:    end for
16:  end for
17:  if  $T(n, p_{max}/2) = \text{false}$  then
18:    Pas de solutions
19:  else
20:    Construire le sous ensemble solution
21:  end if
22: end if

```

Le sous ensemble solution S_{sol} se construit à l'aide du principe suivant : pour une ligne $i > 0$ et un poids $j > 0$ donnés, on a $a_i \in S_{sol}$ si $T(i - 1, j) = \text{FAUX} \wedge T(i, j) = \text{VRAI}$. Ceci n'est vrai que dans un sens et permet d'introduire l'implication suivante :

$$T(i - 1, j) \oplus T(i, j) = \text{VRAI} \Rightarrow a_i \in S_{sol} \quad (1.6)$$

Ceci mérite quelques explications. Séparons les différents cas possibles et présentons les dans un tableau :

$T(i - 1, j) \backslash T(i, j)$	VRAI	FAUX
VRAI	Cas 1	Cas 2
FAUX	Cas 3	Cas 4

Analysons les différents cas :

1. Dans ce cas, l'ensemble des $i - 1$ premiers objets possède un sous-ensemble vérifiant les contraintes imposées. On peut donc en déduire qu'il existe un sous-ensemble ne contenant pas a_i et vérifiant ces mêmes contraintes⁴. C'est cet ensemble qui nous intéresse
2. Ce cas est un cas interdit par la formule de calcul de la ligne i . En effet, si $T(i - 1, j) = \text{VRAI}$, alors $T(i, j) = \text{VRAI}$ par construction.
3. C'est ce cas qui est intéressant, puisqu'il nous indique que l'ajout de l'objet a_i permet de trouver une solution au problème du sous-ensemble pour les valeurs données. On a donc a_i appartenant bel et bien à la solution.

4. Attention, cela ne veut pas dire qu'il n'existe pas de sous-ensembles vérifiant les contraintes et contenant a_i

4. Ce cas indique juste que la solution n'existe pas.

De ces principes, on peut déduire l'algorithme 4 de reconstruction de la solution.

Algorithm 4 Construct-sol

```
1:  $S \leftarrow \emptyset$ 
2: if  $i = 0$  or  $j = 0$  then
3:   Retourne  $S$ 
4: else
5:   if  $T(i - 1, j) = \text{false}$  then
6:     Retourne  $a_i \cup \text{Construct-sol}(i - 1, j - a_i)$ 
7:   else
8:     Retourne  $\text{Construct-sol}(i - 1, j)$ 
9:   end if
10: end if
```

Complexité

$$O(n.p_{max})$$

Jeux d'essais

1.7.2 Le problème du sac à dos

Justification des formules

Exemple

Complexité

1.7.3 Le problème du voyageur de commerce

Exemple

Complexité

1.8 Sur le produit matriciel

1.8.1 Nombre d'opérations nécessaires pour un produit

Soit M_i une (p_i, p_{i+1}) -matrice. Soient π_1 et π_2 les produits matriciels à parenthésage symétrique suivants :

$$\pi_1 = (M_1(M_2(M_3(M_4(\dots(M_{k-1}.M_k))))))$$

$$\pi_2 = ((((((M_1.M_2)M_3)M_4)...)M_{k-1})M_k)$$

Nous admettons que le produit d'une (p, q) -matrice et d'une (q, r) -matrice peut se faire à l'aide de $p.q.r$ opérations. Tous les produits matriciels envisagés par la suite sont définis car le nombre de colonnes de la première matrice est égal au nombre de lignes de la seconde.

Nombre d'opérations nécessaires au produit π_1

Évaluons le nombre d'opérations nécessaires au produit π_1 grâce au tableau suivant :

π_l	$\pi_{1,1}$...	$\pi_{1,4}$...	$\pi_{1,k-2}$	$\pi_{1,k-1}$
=	$M_1 \times$...	$M_4 \times$...	$M_{k-2} \times$	$(M_{k-1} \times M_k)$
<i>Nb. d'opérations</i>	$1.2.(k+1)$...	$4.5.(k+1)$...	$(k-2).(k-1).(k+1)$	$(k-1).k.(k+1)$
<i>Taille de la matrice</i>	$(1, k+1)$...	$(4, k+1)$...	$(k-2, k+1)$	$(k-1, k+1)$

Nous pouvons alors déduire la formule donnant le nombre d'opérations nécessaires pour calculer le produit π_1 :

$$\sum_{i=1}^{i < k} [i \times (i+1) \times (k+1)]$$

Décomposons cette somme :

$$(k+1) \times \left[\sum_{i=1}^{i < k} i^2 + \sum_{i=1}^{i < k} i \right]$$

En remplaçant les sommes par leur valeur :

$$(k+1) \times \left(\frac{k.(k-1).(2k-1)}{6} + \frac{k.(k-1)}{2} \right)$$

En développant :

$$(k+1) \times \left(\frac{2k^3 - 3k^2 + k}{6} + \frac{k^2 - k}{2} \right)$$

En réduisant :

$$\frac{2k^4 + 2k^3 - 2k^2 - 2k}{6}$$

Donc le nombre d'opérations nécessaires pour calculer le produit π_1 est en $O(k^4)$.

Nombre d'opérations nécessaires au produit π_2

De la même façon, évaluons le nombre d'opérations nécessaires au produit π_2 grâce au tableau suivant :

π_l	$\pi_{2,1}$	$\pi_{2,2}$	$\pi_{2,3}$	\dots	$\pi_{2,k-2}$	$\pi_{2,k-1}$
$=$	$M_1 \times M_2$	$\times M_3$	$\times M_4$	\dots	$\times M_{k-1}$	$\times M_k$
<i>Nb. d'operations</i>	1.2.3	1.3.4	1.4.5	\dots	$1.(k-1).k$	$1.k.(k+1)$
<i>Taille de la matrice</i>	(1, 3)	(1, 4)	(1, 5)	\dots	(1, k)	(1, k + 1)

Nous pouvons alors d  duire la formule donnant le nombre d'op  rations n  cessaires pour calculer le produit π_2 :

$$\sum_{i=1}^{i < k} [(i+1) \times (i+2)]$$

D  composons cette somme :

$$2k - 2 + \sum_{i=1}^{i < k} i^2 + 3 \times \sum_{i=1}^{i < k} i$$

En rempla  ant les sommes par leur valeur :

$$2k - 2 + \frac{k.(k-1).(2k-1)}{6} + 3 \times \frac{k.(k-1)}{2}$$

En d  veloppant :

$$\frac{12k - 12}{6} + \frac{2k^3 - 3k^2 + k}{6} + \frac{9k^2 - 9k}{6}$$

En r  duisant :

$$\frac{2k^3 + 6k^2 + 4k - 12}{6}$$

Donc le nombre d'op  rations n  cessaires pour calculer le produit π_2 est en $O(k^3)$.

Comparaison

Comparons maintenant le nombre d'op  rations n  cessaires aux produits π_1 et π_2 :

	π_1	<i>vs</i>	π_2
$k = 2$	6	$=$	6
$k = 3$	32	$>$	18
$k = 4$	100	$>$	38

Le premier crit  re de comparaison est le fait que le nombre d'op  rations n  cessaires pour calculer le produit π_1 est en $O(k^4)$ alors que le nombre d'op  rations n  cessaires pour calculer le produit π_2 est en $O(k^3)$. De fa  on   vidence, π_2 est plus int  ressant car il co  te k fois moins d'op  rations.

D'autre part, nous déduisons du tableau précédent que pour deux matrices, le nombre d'opérations est le même (en effet, il n'y a qu'une seule manière de multiplier deux matrices). Par contre pour tout nombre de matrices strictement supérieur à 2, π_2 est plus intéressant.

Le nombre d'opérations nécessaires pour calculer un produit matriciel dépendant donc du parenthésage choisi.

1.8.2 Nombre de parenthésages possibles d'un produit de k matrices

Montrons que le nombre de parenthésages possibles est le suivant : $c(k) = \sum_{i=1}^{k-1} c(i)c(k-i)$.
Posons $c(1) = 1$.

Pour $n = 2$, il n'y a qu'une façon de parenthéser le produit de deux matrices donc $c(2) = 1$. Or, $c(1).c(2-1) = 1 \times 1 = 1$. Donc la récurrence est initialisée.

Nous admettrons que $\forall k \in \mathbb{N}$ on a : $c(k) = \sum_{i=1}^{k-1} c(i)c(k-i)$.

Pour $k = n + 1$, on a les mêmes façons de parenthéser le produit que pour $k = n$ plus .

1.9 Résolution numérique par la méthode primal-dual

Jocker !

1.10 Seuil d'approximation pour le problème Bin Packing

1.10.1 Définition du problème

Soient un ensemble $O = \{o_i : i \in [1; n]\}$ de n objets, une fonction $q : O \rightarrow \mathbb{N}$, une taille de boîte Q .

Le problème de décision Bin Packing est le suivant :
Existe-t-il une k -partition $O_1, \dots, O_k \subseteq O$ telle que :

- $O = \cup_{i=1}^k O_i$,
- $\forall i, j \in \mathbb{N}, i \neq j, O_i \cap O_j = \emptyset$,
- $\sum_{o_{ij} \in O_i} q(o_{ij}) \leq Q \forall i \in [1; k]$.

Le problème d'optimisation associé est :

Minimiser $k \in \mathbb{N} : \exists k$ -partition qui respecte les conditions ci-dessus.

Dans la suite on notera $B(O_i) = \sum_{o_{ij} \in O_i} q(o_{ij})$ la somme des poids des éléments de O_i ,
 $B = B(O)$, et K^* la solution optimale au problème d'optimisation.

1.10.2 Construction d'une instance de Bin Packing à partir d'une instance de Partition

Remarque : pour les définitions et notations concernant le problème de partition, voir l'exercice 1.5.

Soit x une instance du problème partition, nous construisons une instance x' de Bin Packing de la manière suivante :

- $O = A$,
- $q(o_i) = \frac{2 \times p(a_i)}{w(A)}$,
- $Q = 1$.

1.10.3 Instance positive

Nous devons montrer que si x est positive alors $K^*(x') = 2$ et que $K^*(x') = 3$ sinon. Commençons par calculer $B = \sum_{a_i \in A} \frac{2 \times p(a_i)}{w(A)} = \frac{2}{w(A)} \times \sum_{a_i \in A} p(a_i) = \frac{2}{w(A)} \times w(A) = 2$.

Montrons tout d'abord que $K^*(x') \leq 3$.

Nous supposons que $K(x') = 4 > 3$. Nous savons alors qu'il existe quatre ensembles O_1, O_2, O_3, O_4 tels que $B(O_i) \leq 1$. De plus, puisque les O_i sont disjoints, nous avons $B = B(O_1) + B(O_2) + B(O_3) + B(O_4)$, et nous pouvons énumérer les O_i de sorte que $B(O_i) \geq B(O_{i+1})$.

Or, $B(O_i) + B(O_j) > 1 \forall i \neq j$, en effet si ce n'était pas le cas, alors on pourrait unir les deux ensembles et $K^*(x')$ serait égal à 3. En particulier nous avons : $B(O_1) + B(O_2) > 1$ et $B(O_3) + B(O_4) > 1$, c'est à dire : $2 = B(O_1) + B(O_2) + B(O_3) + B(O_4) > 2$, ce qui est absurde.

Nous avons donc que $K(x') \leq 3$.

Supposons que x soit une instance positive.

Soit la solution P de x Y_1, Y_2 , on a donc $w(Y_1) = w(Y_2) = \frac{w(A)}{2} = P$. Soit O_1 (resp. O_2) l'image de Y_1 (resp. Y_2) dans le problème x' .

On a donc : $B(O_2) = B(O_1) = \frac{2 \times w(Y_1)}{w(A)} = \frac{2 \times w(A)}{2 \times w(A)} = 1 \leq Q$.

Ainsi $K^* = 2$.

Supposons maintenant que x soit une instance négative et on suppose que $K(x') < 3$.

Si $K(x') = 1$, il est trivial de voir que $B = 2$ et que $B \leq Q = 1$ ce qui est absurde.

Si $K(x') = 2$, on a alors $O = O_1 \cup O_2$. De plus, $B = 2$, $B = B(O_1) + B(O_2)$ et $B(O_i) \leq 1$.

On a donc $B(O_1) = B(O_2) = 1$.

Soit Y_1 (resp. Y_2) l'ensemble correspondant à O_1 (resp. O_2) dans le problème x . On a : $1 = w(O_1) = \frac{2}{w(A)} \times w(Y_1) \leftrightarrow w(Y_1) = \frac{w(A)}{2}$. Donc il existe une solution Y_1, Y_2 au problème x , ce qui est absurde.

Ainsi si l'instance x est négative, $K^*(x') = 3$.

1.10.4 Classe d'approximation du Bin Packing

Supposons qu'il existe un algorithme A qui donne une solution $(\frac{3}{2} - \epsilon)$ -approchée au problème Bin Packing.

Soit x un problème de Partition. Grâce à la réduction polynomiale étudiée ci-dessus, nous avons donc : $sol_A(x') < \frac{3}{2} sol_{opt}(x')$.

Et si le problème partition admet une solution alors $sol_{opt}(x') = 2$, nous en déduisons : $sol_A(x') < 3$, ie $sol_A(x') = 2 = sol_{opt}(x')$, et si ce n'est pas le cas $sol_A(x') = 3$. A permet donc de résoudre le problème Partition en temps polynomial, celui-ci étant NP-complet, il n'existe pas un tel algorithme.

Ainsi le problème Bin Packing admet un seuil d'approximation en $\frac{3}{2}$ et donc n'appartient pas à la classe d'approximation *PTAS*.

1.11 Seuil d'approximation pour le problème de la coloration de sommets (resp. d'arêtes)

Soit un graphe quelconque $G = (V, E)$.

Coloration de sommets

Le problème de décision est le suivant :

soit $k \in \mathbb{N}$, est-il possible de trouver une fonction $\chi : V \rightarrow [1, k]$ telle que $\forall (x, y) \in E, \chi(x) \neq \chi(y)$.

Son problème d'optimisation associé est de trouver k le plus petit possible tel que G soit k -colorable (on note $\chi(G)$ un tel k).

Coloration d'arêtes

Le problème de décision est le suivant :

soit $k \in \mathbb{N}$, est-il possible de trouver une fonction $\chi' : E \rightarrow [1, k]$ telle que $\forall (x, y_1), (x, y_2) \in E, \chi'((x, y_1)) \neq \chi'((x, y_2))$.

Son problème d'optimisation associé est de trouver k le plus petit possible tel que G soit k -arêtes-colorable (on note $\chi'(G)$ un tel k).

1.11.1 Complexité

Ces deux problèmes de coloration appartiennent à la classe *NP*, de plus, celui de sommets est non approximable, tandis que celui d'arêtes l'est mais n'appartient pas à la classe d'approximation *PTAS*.

Nous nous intéressons à déterminer un seuil d'approximation pour le problème de coloration d'arêtes.

1.11.2 Algorithme approché avec performance relative meilleure que $\frac{4}{3}$

Supposons qu'il existe un algorithme $(\frac{4}{3} - \epsilon)$ -approché pour le problème de coloration des arêtes. Nous avons alors $sol_A < \frac{4}{3} sol_{opt}$. De plus, ce problème d'optimisation étant un problème de minimisation, nous avons $sol_A \geq sol_{opt}$.

(a)

Si $sol_{opt} \leq 3$, on a : $sol_A < \frac{4}{3} \times sol_{opt} \leq \frac{4}{3} \times 3 = 4$, c'est à dire $sol_A < 4$, et $sol_A \in \mathbb{N}$, ie $sol_A \leq 3$.

Cet algorithme renvoie donc la solution optimale.

(b)

Si $sol_{opt} \geq 4$, l'algorithme renverra une solution strictement supérieure à 3. Ainsi il est possible de déterminer si la solution optimale est inférieure à 3.

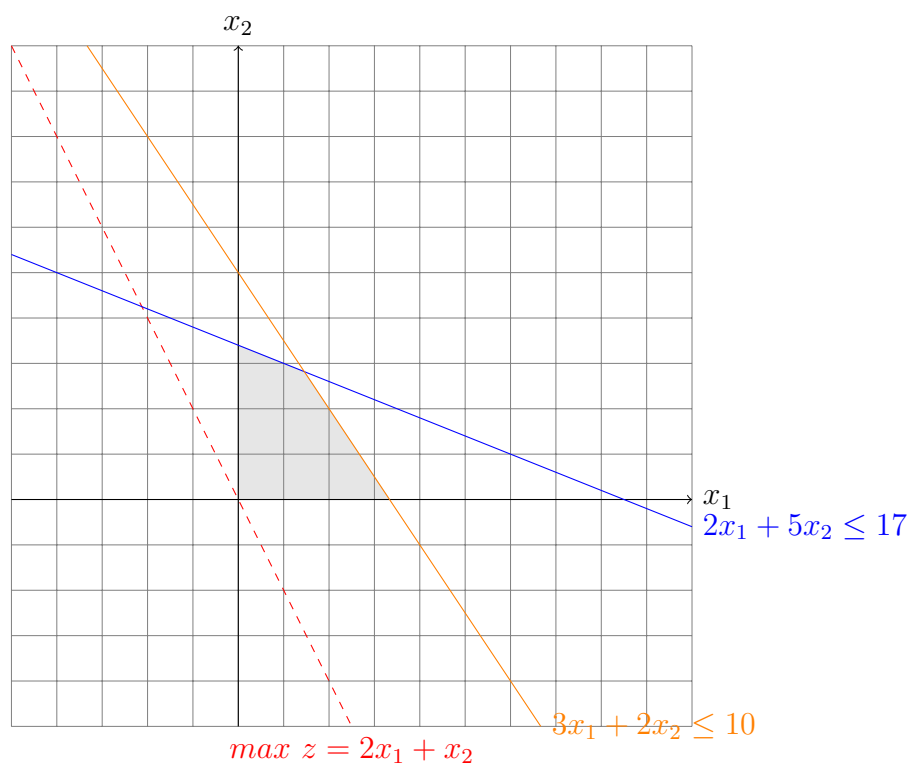
1.11.3 Conclusion

Etant donné que si un algorithme de rapport strictement inférieur à $\frac{4}{3}$ existait, nous pourrions répondre en temps polynomial au problème de savoir si un graphe est 3-arêtes-colorable (qui est NP-complet), le seuil d'approximation minimum pour le problème de coloration d'arêtes est donc supérieur ou égal à $\frac{4}{3}$.

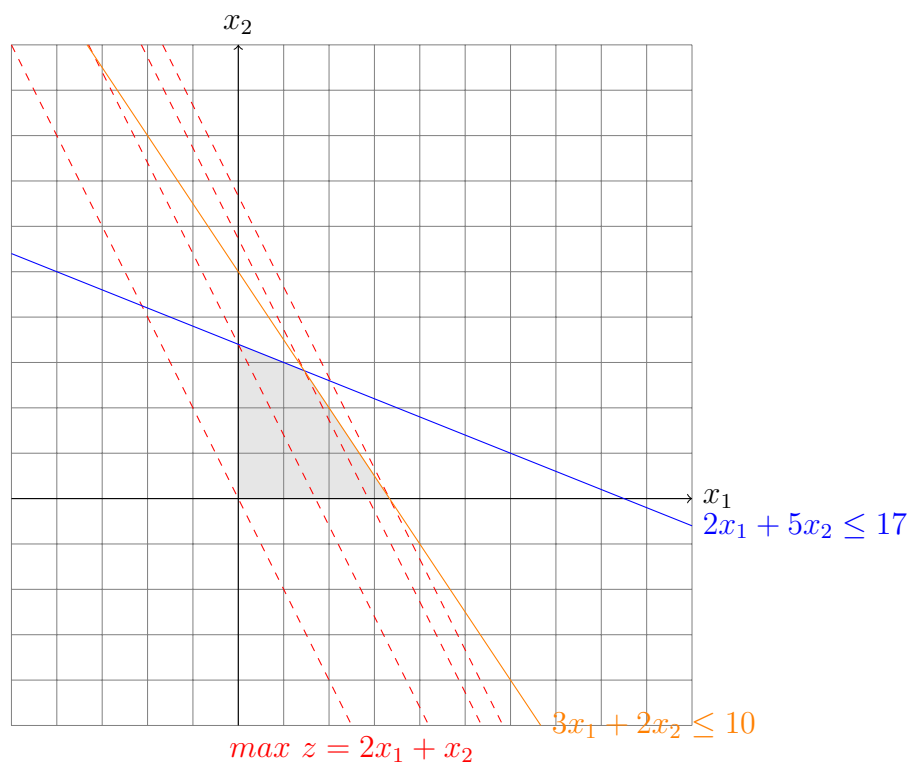
1.12 Comparaison de *branch and bound* et *branch and cut*

Nous considérons le programme linéaire suivant :

$$PL_0 \begin{cases} \max z(x_1, x_2) = 2x_1 + x_2 \\ 2x_1 + 5x_2 \leq 17 \\ 3x_1 + 2x_2 \leq 10 \\ x_1, x_2 \geq 0 \end{cases}$$

1.12.1 Polytope associé à PL_0 

1.12.2 Résolution graphique



Nous obtenons grâce à la résolution graphique :

$$\begin{cases} x_1 = \frac{10}{3} \\ x_2 = 0 \\ z = \frac{20}{3} \end{cases}$$

1.12.3 Résolution par la méthode du simplexe

Programme linéaire :

$$PL_0 \begin{cases} \max z(x_1, x_2) = 2x_1 + x_2 \\ 2x_1 + 5x_2 \leq 17 \\ 3x_1 + 2x_2 \leq 10 \\ x_1, x_2 \geq 0 \end{cases}$$

Forme standard :

$$PL_0 \begin{cases} \max z(x_1, x_2) = 2x_1 + x_2 \\ 2x_1 + 5x_2 + y_1 = 17 \\ 3x_1 + 2x_2 + y_2 = 10 \\ x_1, x_2, y_1, y_2 \geq 0 \end{cases}$$

Tableaux :

		2	1	0	0
0	$y_1 = 17$	2	5	1	0
0	$y_2 = 10$	3	2	0	1
	$z = 0$	-2	-1	0	0

y_2 sort de la base ; x_1 rentre dans la base ; le pivot devient $a_{1,2} = 3$.

		2	1	0	0
0	$y_1 = \frac{31}{3}$	0	$\frac{11}{3}$	1	$-\frac{2}{3}$
2	$x_1 = \frac{10}{3}$	1	$\frac{2}{3}$	0	$\frac{1}{3}$
	$z = \frac{20}{3}$	0	$\frac{1}{3}$	0	$\frac{2}{3}$

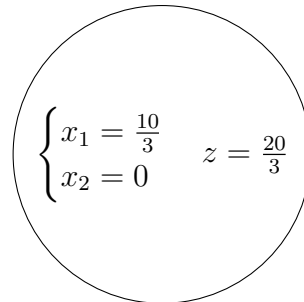
Tous les coûts réduits sont positifs ou nuls donc le simplexe est fini. La solution correspond bien à celle trouvée graphiquement :

$$\begin{cases} x_1 = \frac{10}{3} \\ x_2 = 0 \\ z = \frac{20}{3} \end{cases}$$

1.12.4 Recherche d'une solution à valeur entière

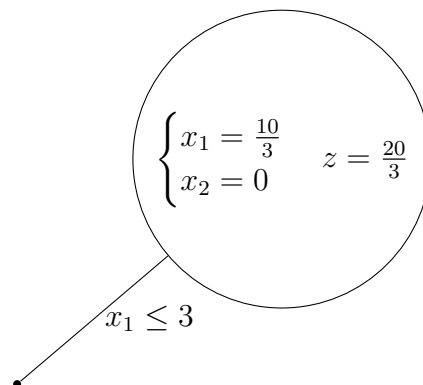
Branch and bound

Pour la méthode du Branch and bound, nous partons de la solution optimale réelle trouvée précédemment.



$$\begin{cases} x_1 = \frac{10}{3} \\ x_2 = 0 \end{cases} \quad z = \frac{20}{3}$$

Puis nous ajoutons la contrainte $x_1 \leq 3$ pour forcer x_1 à être entier.



$$\begin{cases} x_1 = \frac{10}{3} \\ x_2 = 0 \end{cases} \quad z = \frac{20}{3}$$

$x_1 \leq 3$

Nous calculons alors les tableaux du simplexe avec la nouvelle contrainte.

		2	1	0	0	0
0	$y_1 = 17$	2	5	1	0	0
0	$y_2 = 10$	3	2	0	1	0
0	$y_3 = 3$	1	0	0	0	1
	$z = 0$	-2	-1	0	0	0

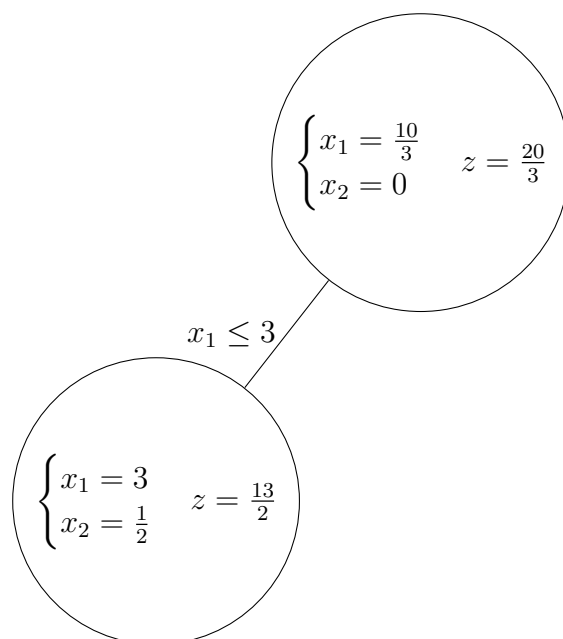
y_3 sort de la base ; x_1 rentre dans la base ; le pivot devient $a_{1,3} = 1$.

		2	1	0	0	0
0	$y_1 = 11$	0	5	1	0	-2
0	$y_2 = 1$	0	2	0	1	-3
2	$x_1 = 3$	1	0	0	0	1
	$z = 6$	0	-1	0	0	2

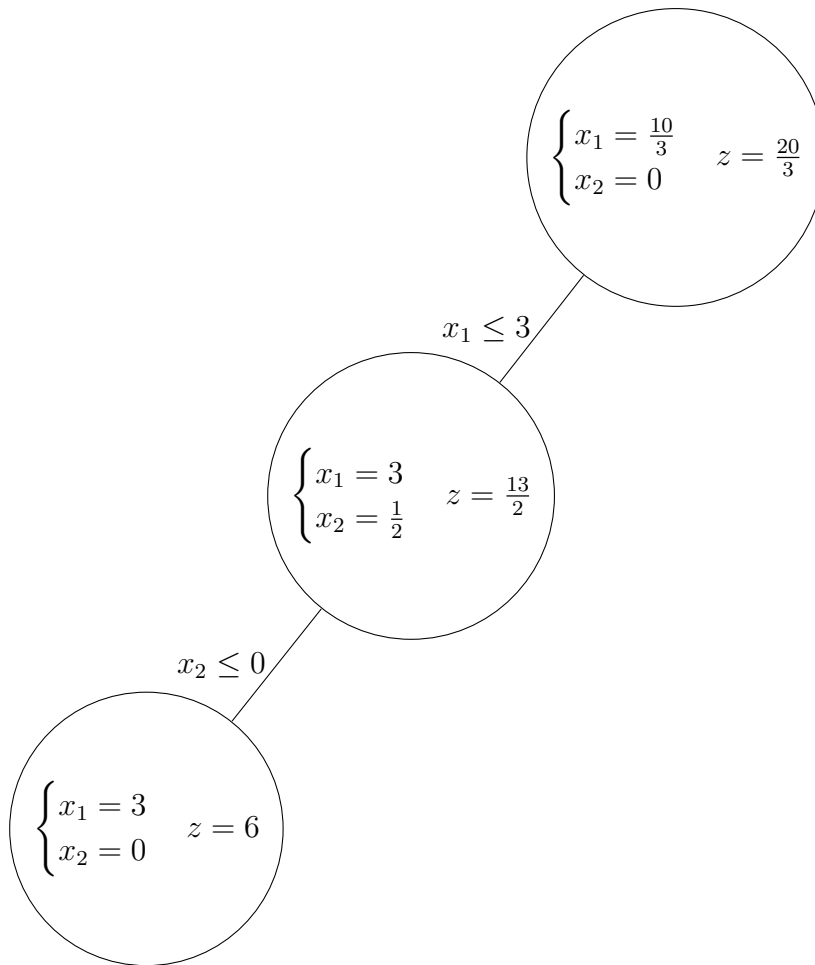
y_2 sort de la base ; x_2 rentre dans la base ; le pivot devient $a_{2,2} = 2$.

		2	1	0	0	0
0	$y_1 = \frac{17}{2}$	0	0	1	$-\frac{5}{2}$	2
1	$x_2 = \frac{1}{2}$	0	1	0	$\frac{1}{2}$	$-\frac{3}{2}$
2	$x_1 = 3$	1	0	0	0	1
	$z = \frac{13}{2}$	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$

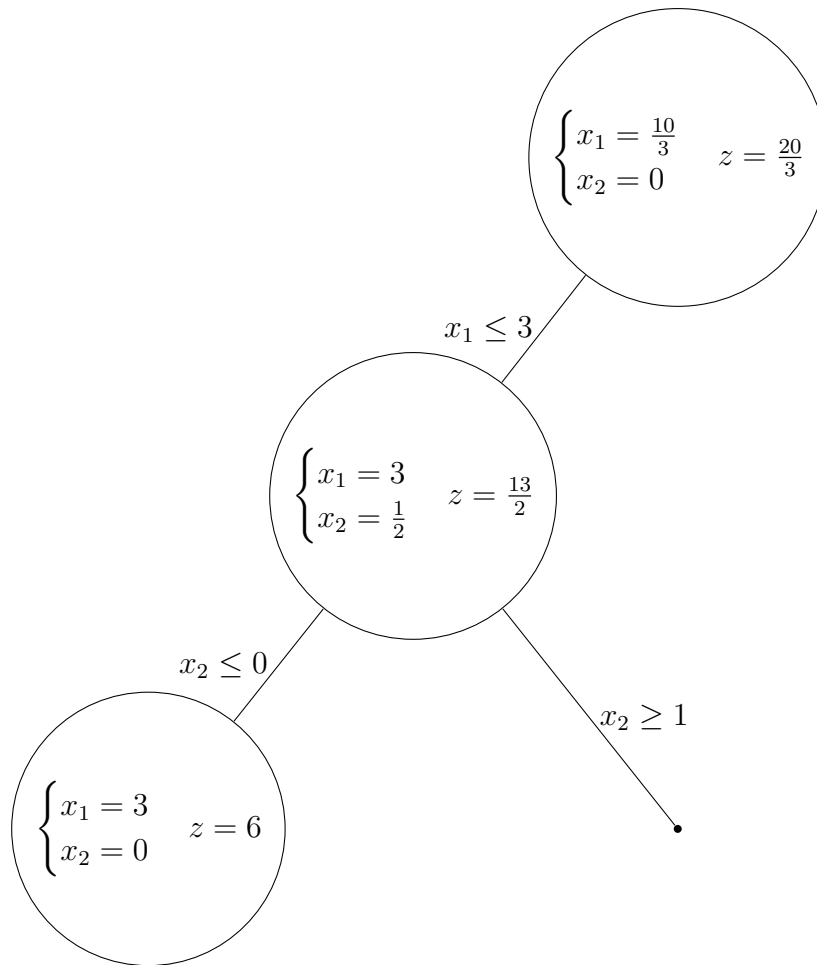
Le simplexe est fini, nous ajoutons le résultat à l'arbre de branch and cut.



x_1 a bien une valeur entière mais x_2 est devenu réel, nous ajoutons donc la contrainte $x_2 \leq 0$. Comme $x_2 \geq 0$ par définition, nous obtenons la solution suivante :



Nous obtenons une feuille de l'arbre et une solution entière admissible. Nous devons maintenant développer le reste de l'arbre pour vérifier s'il est possible d'obtenir une meilleure solution entière. Posons donc la contrainte suivante : $x_2 \geq 1$.



Afin de trouver une solution, nous appliquons la phase 1 du simplexe dont voici les tableaux.

		0	0	0	0	0	0	-1
0	$y_1 = 17$	2	5	1	0	0	0	0
0	$y_2 = 10$	3	2	0	1	0	0	0
0	$y_3 = 3$	1	0	0	0	1	0	0
-1	$y_5 = 1$	0	1	0	0	0	-1	1
	$z = -1$	0	-1	0	0	0	1	0

y_5 sort de la base ; x_2 rentre dans la base ; le pivot devient $a_{2,4} = 1$.

		0	0	0	0	0	0	-1
0	$y_1 = 12$	2	0	1	0	0	5	-5
0	$y_2 = 8$	3	0	0	1	0	2	-2
0	$y_3 = 3$	1	0	0	0	1	0	0
0	$x_2 = 1$	0	1	0	0	0	-1	1
	$z = 0$	0	0	0	0	0	0	1

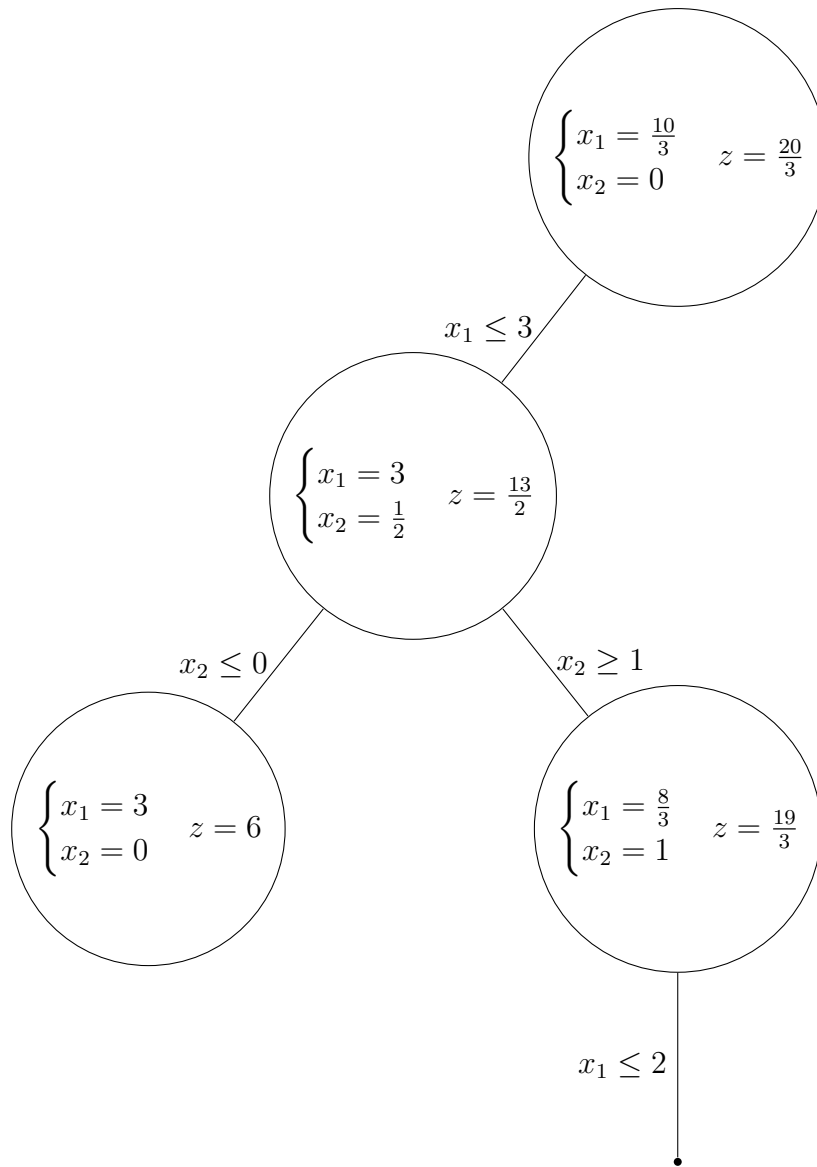
$z = 0$ donc la phase 1 du simplexe est finie. Passons à la phase 2...

		2	1	0	0	0	0
0	$y_1 = 12$	2	0	1	0	0	5
0	$y_2 = 8$	3	0	0	1	0	2
0	$y_3 = 3$	1	0	0	0	1	0
1	$x_2 = 1$	0	1	0	0	0	-1
	$z = 1$	-2	0	0	0	0	-1

y_2 sort de la base ; x_1 rentre dans la base ; le pivot devient $a_{1,2} = 3$.

		2	1	0	0	0	0
0	$y_1 = \frac{20}{3}$	0	0	1	$-\frac{2}{3}$	0	$\frac{11}{3}$
2	$x_1 = \frac{8}{3}$	1	0	0	$\frac{1}{3}$	0	$\frac{2}{3}$
0	$y_3 = \frac{1}{3}$	0	0	0	$-\frac{1}{3}$	1	$-\frac{2}{3}$
1	$x_2 = 1$	0	1	0	0	0	-1
	$z = \frac{19}{3}$	0	0	0	$\frac{2}{3}$	0	$\frac{1}{3}$

Nous ajoutons la solution obtenue à l'arbre de branch and cut. Comme dans cette solution, x_1 n'est pas entier, nous ajoutons également la contrainte : $x_1 \leq 2$.



Nous lançons alors la résolution de ce nouveau problème par le simplexe.

		2	1	0	0	0	0
0	$y_1 = 12$	2	0	1	0	0	5
0	$y_2 = 8$	3	0	0	1	0	2
0	$y_3 = 2$	1	0	0	0	1	0
1	$x_2 = 1$	0	1	0	0	0	-1
	$z = 1$	-2	0	0	0	0	-1

y_3 sort de la base ; x_1 rentre dans la base ; le pivot devient $a_{1,3} = 1$.

		2	1	0	0	0	0
0	$y_1 = 8$	0	0	1	0	-2	5
0	$y_2 = 2$	0	0	0	1	-3	2
2	$x_1 = 2$	1	0	0	0	1	0
1	$x_2 = 1$	0	1	0	0	0	-1
	$z = 3$	0	0	0	0	2	-1

y_2 sort de la base ; y_4 rentre dans la base ; le pivot devient $a_{6,2} = 2$.

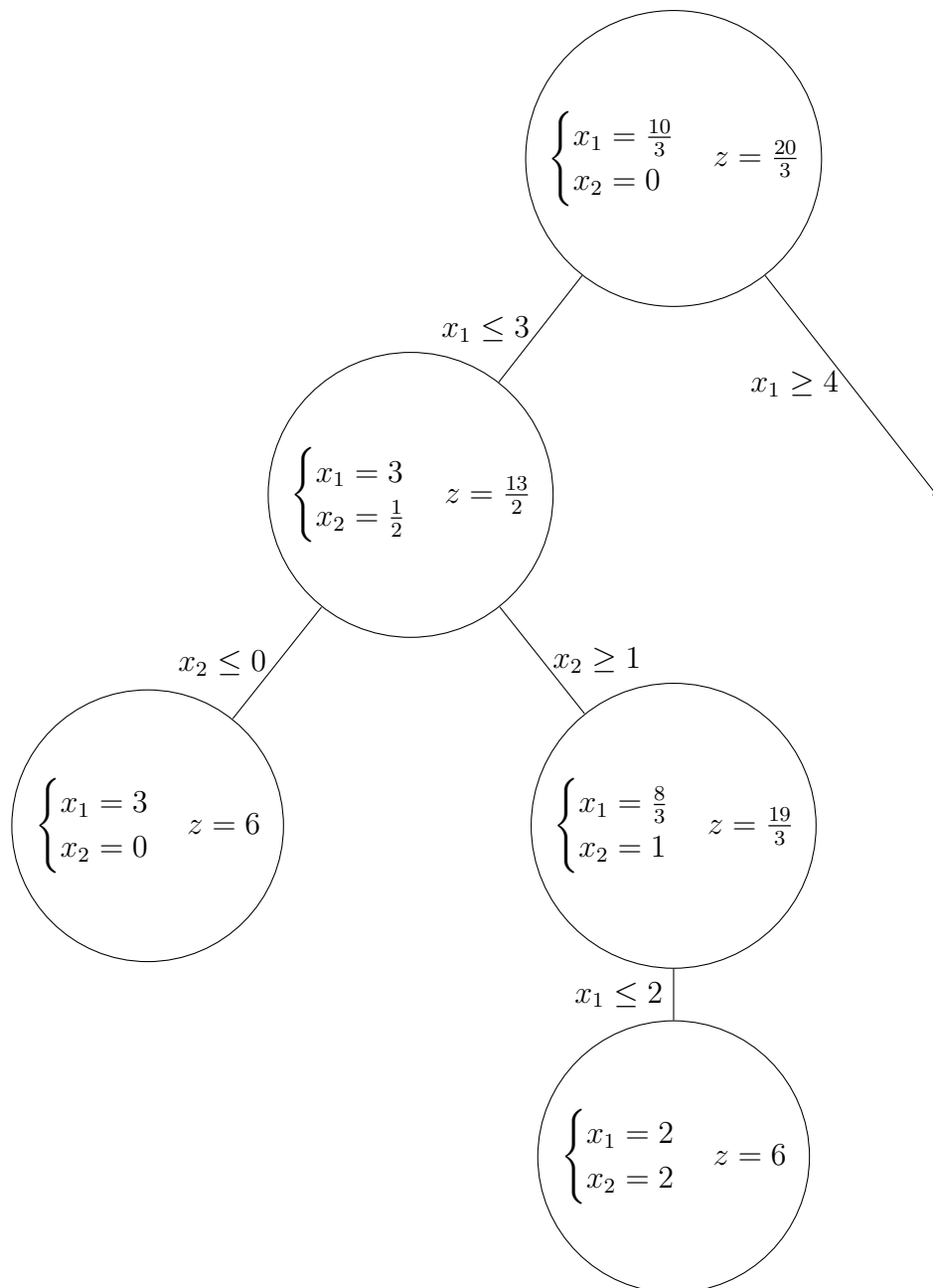
		2	1	0	0	0	0
0	$y_1 = 3$	0	0	1	$-\frac{5}{2}$	$\frac{11}{2}$	0
0	$y_4 = 1$	0	0	0	$\frac{1}{2}$	$-\frac{3}{2}$	1
2	$x_1 = 2$	1	0	0	0	1	0
1	$x_2 = 2$	0	1	0	$\frac{1}{2}$	$-\frac{3}{2}$	0
	$z = 6$	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	0

La résolution par le simplexe est finie et nous obtenons une solution entière égale à celle précédemment trouvée. Cette solution est donc une feuille de l'arbre ; nous remontons d'un niveau et ajoutons la contrainte $x_1 \geq 4$.

Nous remarquons que cette contrainte force x_1 à violer la deuxième contrainte du problème originel : $3x_1 + 2x_2 \leq 10$. En effet $3 \times 4 > 10$, cette contrainte ne mène donc à aucune nouvelle solution et l'arbre de branch and cut est terminé.

Nous avons donc trouvé deux solutions optimales entières au problème :

$$\begin{cases} x_1 = 2 \\ x_2 = 2 \\ z = 6 \end{cases} \quad \text{et} \quad \begin{cases} x_1 = 3 \\ x_2 = 0 \\ z = 6 \end{cases}$$



Coupes de Gomory

La méthode des coupes de Gomory nécessite une solution optimale réelle, nous partons donc de celle trouvée précédemment :

$$\begin{cases} x_1 = \frac{10}{3} \\ x_2 = 0 \\ z = \frac{20}{3} \end{cases}$$

et du dernier tableau du simplexe associé :

		2	1	0	0
0	$y_1 = \frac{31}{3}$	0	$\frac{11}{3}$	1	$-\frac{2}{3}$
2	$x_1 = \frac{10}{3}$	1	$\frac{2}{3}$	0	$\frac{1}{3}$
	$z = \frac{20}{3}$	0	$\frac{1}{3}$	0	$\frac{2}{3}$

Nous avons choisi la deuxième ligne du tableau pour en déduire la contrainte de Gomory suivante :

$$\begin{aligned} < \frac{2}{3} > x_2 + < \frac{1}{3} > y_2 \geq < \frac{10}{3} > \\ \Leftrightarrow \\ \frac{2}{3}x_2 + \frac{1}{3}y_2 \geq \frac{10}{3} \end{aligned}$$

Nous devons maintenant l'exprimer uniquement en fonction de x_1 et x_2 . Remplaçons donc y_2 par son équivalent dans la contrainte 2, cela donne :

$$x_1 \leq 3$$

Nous devons maintenant utiliser le simplexe pour résoudre le problème avec la nouvelle contrainte. L'ayant déjà fait lors de la question précédente, sautons directement au résultat :

$$\begin{cases} x_1 = 3 \\ x_2 = \frac{1}{2} \\ z = \frac{13}{2} \end{cases}$$

et le dernier tableau associé :

		2	1	0	0	0
0	$y_1 = \frac{17}{2}$	0	0	1	$-\frac{5}{2}$	2
1	$x_2 = \frac{1}{2}$	0	1	0	$\frac{1}{2}$	$-\frac{3}{2}$
2	$x_1 = 3$	1	0	0	0	1
	$z = \frac{13}{2}$	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$

Nous choisissons la ligne 2 du tableau pour exprimer la contrainte de Gomory :

$$\begin{aligned} < \frac{1}{2} > y_2 + < -\frac{3}{2} > y_3 \geq < \frac{1}{2} > \\ \Leftrightarrow \\ \frac{1}{2}y_2 + \frac{1}{2}y_3 \geq \frac{1}{2} \end{aligned}$$

Nous devons maintenant l'exprimer uniquement en fonction de x_1 et x_2 . Remplaçons donc y_2 par son équivalent dans la contrainte 2, cela donne :

$$x_2 \leq 2y_3$$

Remplaçons maintenant y_3 par son équivalent dans la contrainte 3, cela donne :

$$2x_1 + x_2 \leq 6$$

Réolvons le problème avec cette contrainte en plus :

		2	1	0	0	0	0
0	$y_1 = 17$	2	5	1	0	0	0
0	$y_2 = 10$	3	2	0	1	0	0
0	$y_3 = 3$	1	0	0	0	1	0
0	$y_4 = 6$	2	1	0	0	0	1
	$z = 0$	-2	-1	0	0	0	0

y_4 sort de la base ; x_1 rentre dans la base ; le pivot devient $a_{1,4} = 2$.

		2	1	0	0	0	0
0	$y_1 = 11$	0	4	1	0	0	-1
0	$y_2 = 1$	0	$\frac{1}{2}$	0	1	0	$-\frac{3}{2}$
0	$y_3 = 0$	0	$-\frac{1}{2}$	0	0	1	$-\frac{1}{2}$
2	$x_1 = 3$	1	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$
	$z = 6$	0	0	0	0	0	1

Et voici donc la solution finale de la méthode des coupes de Gomory : $\begin{cases} x_1 = 3 \\ x_2 = 0 \\ z = 6 \end{cases}$.

Chapitre 2

Partie pratique

2.1 Programmation dynamique

Nous avons décidé d'implémenter tous les algorithmes suivants en langage C pour des raisons d'efficacité.

2.1.1 Problème de la partition

Description du problème

Le problème de la 2-partition de n objets consiste à trouver deux sous-ensembles de ces n objets tel que chacun des sous-ensembles ait le même poids. Les poids de chacun des objets étant des valeurs entières, pour qu'il existe deux sous-ensembles distincts ayant le même poids, il faut que la somme des poids des objets soit paire.

Algorithme de programmation dynamique

Prenons l'algorithme de résolution présenté dans la section théorique (cf 3) et rappelé ci-dessous. Cet algorithme est un algorithme exact admettant une complexité en $O(n.P)$ avec n le nombre d'objets et P la somme des poids de tous les objets.

Algorithm 5 solve-partition

```
1:  $P \leftarrow \sum_{i \in [1, n]} p(a_i)$ 
2: if  $P \equiv 1 \pmod{2}$  then
3:   Pas de solutions
4: else
5:   for  $j \in [0, P/2]$  do
6:     if  $j = 0$  then
7:        $T(0, j) \leftarrow \text{true}$ 
8:     else
9:        $T(0, j) \leftarrow \text{false}$ 
10:    end if
11:  end for
12:  for  $i \in [1, n]$  do
13:    for  $j \in [0, P/2]$  do
14:       $T(i, j) \leftarrow [(j = 0) \vee (j = p(a_i)) \vee (T(i - 1, j)) \vee (T(i - 1, j - p(a_i)))]$ 
15:    end for
16:  end for
17:  if  $T(n, P/2) = \text{false}$  then
18:    Pas de solutions
19:  else
20:    Construire le sous ensemble solution
21:  end if
22: end if
```

Tests

2.1.2 Problème du sac à dos

Description du problème

Le problème du sac à dos consiste à vouloir mettre des objets ayant chacun différents volumes et utilités dans un sac à dos à volume limité. Le but est d'obtenir un sac à dos rempli avec une utilité maximum.

Algorithme de programmation dynamique

Prenons l'algorithme de résolution présenté ci-dessous. Cet algorithme est un algorithme exact admettant une complexité en $O(n.V^2)$ avec n le nombre d'objets et V le volume du sac à dos. Notons que la borne (pire des cas) est atteinte si tous les objets sont de volume 1.

Algorithm 6 sac à dos

```

1: for  $j \in \{1, \dots, volumeMax\}$  do
2:    $T[0, j] \leftarrow 0$ 
3: end for
4: for  $i \in \{1, \dots, n\}$  do
5:   for  $j \in \{1, \dots, volumeMax\}$  do
6:     if  $j \neq 1$  then
7:        $T[i, j] \leftarrow T[i, j - 1]$ 
8:     end if
9:     for  $k \in \{0, \dots, volumeMax/volume[i]\}$  do
10:       $T[i, j] \leftarrow \max(T[i, j], T[i - 1, j - k \times volume[i]] + k \times utilite[i])$ 
11:    end for
12:   end for
13: end for
14: return  $T[n, volumeMax]$ 

```

Tests**2.1.3 Problème du voyageur de commerce****Description du problème**

Le problème du voyageur de commerce consiste à trouver un cycle hamiltonien de poids minimum.

Algorithme de programmation dynamique**Algorithm 7** TSP

```

1: for  $i \in \{1, \dots, n - 1\}$  do
2:    $MeilleureChaine(\{v_i\}, v_i) = (v_0, v_i)$ 
3:    $ValeurChaine(\{v_i\}, v_i) = d(v_0, v_i)$ 
4: end for
5: for  $j \in \{1, \dots, n - 2\}$  do
6:   for  $V' \subseteq \{v_1, \dots, v_{n-1}\} : |V'| = j$  do
7:     for  $v \in V'$  do
8:        $ValeurChaine(V', v) = \min_{w \in V'} \{ValeurChaine(V' \setminus \{v\}, w) + d(w, v)\}$ 
9:        $MeilleureChaine(V', v) = (MeilleureChaine(V' \setminus \{v\}, w_0), v)$  (où  $w_0$  est le sommet qui atteint ce minimum)
10:    end for
11:   end for
12: end for
13: return  $T = \arg \min_{v=v_2}^{v_n} (ValeurChaine(\{v_1, \dots, v_{n-1}\} \setminus \{v\}, v) + d(v, v_0))$ 

```

2.2 Branch and bound

2.2.1 Principe général du branch&bound

La méthode du branch and bound consiste à énumérer les solutions d'un problème sous forme d'arbre binaire et à conserver à chaque pas la meilleure solution réalisable. Certaines branches de l'arbre ne sont pas évaluées lorsqu'elles donnent de moins bonnes solutions que la meilleure solution courante. Une méthode de séparation est appliquée à chaque nœud de l'arbre pour le partitionner en deux sous-problèmes et ainsi générer ses fils. Une méthode d'évaluation, quant à elle, sert à déterminer la solution optimale associée à un nœud de l'arbre.

2.2.2 Méthode de séparation

2.2.3 Méthode d'évaluation

2.2.4 Solution initiale du TSP

Chaîne de poids le plus faible

voisinage 2-opt

voisinage 3-opt

2.2.5 Comparaisons

2.2.6 Algorithme $\frac{3}{2}$ approché

2.3 Comparaison du Branch&Bound et de la programmation dynamique sur l'exemple du TSP