Chapitre 9 : Schémas algorithmiques avancés

1 Algorithmes d'approximation

1.1 Rappels : problèmes d'optimisation

Définition 1.1

Soit $Q \subseteq \mathcal{E} \times \mathcal{S}$ un problème. Soit $\text{opt} \in \{\min, \max\}$.
*On dit que Q est un **problème d'**opt **imisation** si pour toute entrée $e \in \mathcal{E}$ il existe :*
- *un ensemble $\textit{sol}(e)$,*
- *une fonction $c_e \in \textit{sol}(e) \to \mathbb{R}^+$,*

tels que $c_e^\star = \text{opt}\{c_e(s) \mid s \in \textit{sol}(e)\}$ est bien défini et $\forall s \in \textit{sol}(e),\ (e, s) \in Q \Rightarrow c_e(s) = c_e^\star$.

Vocabulaire 1.2

Pour une instance $e \in \mathcal{E}$ fixée, on utilise le vocabulaire suivant :
- *$\textit{sol}(e)$ est appelé l'**ensemble des solutions** ;*
- *c_e est appelé la **fonction objectif** ;*
- *$c^\star$ la **valeur** opt **imale** ;*
- *pour une solution $s \in \textit{sol}(e)$, $c_e(s)$ est appelée la **valeur d'une solution**.*

Exemple 1.3

Le problème du plus court chemin dans un graphe connexe.

$$\begin{cases} \textbf{Entrée :} \text{ Un graphe orienté pondéré } G = (S, A, w)\text{, deux sommets } s \text{ et } t \text{ de } S \\ \textbf{Sortie :} \text{ La longueur d'un plus court chemin de } s \text{ à } t \end{cases}$$

L'ensemble des solutions associées à l'entrée $G = (S, A, w)$, s et t est l'ensemble des chemins de s à t dans G. La valeur (la fonction objectif) d'une solution (d'un chemin de s à t) est alors la longueur du chemin pour la pondération w.

Vocabulaire 1.4

*Bien que dans le cadre d'un problème quelconque $Q \subseteq \mathcal{E}_Q \times \mathcal{S}_Q$ on ait appelé solution d'une instance $e \in \mathcal{E}_Q$ tout élément $s \in \mathcal{S}_Q$ tel que $(e, s) \in Q$, dans le cadre des problèmes d'optimisation, ce n'est pas la valeur optimale pour une instance qu'on appelle solution. Comme précisé ci-dessus, pour une instance donnée, on appelle **solution** tout élément de l'ensemble sur lequel on optimise. Notamment, ce terme n'est pas réservé aux éléments réalisant la valeur optimale.*
Pour le problème de plus court chemin par exemple, pour une instance (G, s, t) donnée, on appelle solution n'importe quel chemin de s à t dans G, et valeur optimale la distance de s à t dans G.

Remarque 1.5

Attention, on peut dire **la** valeur optimale, car il y a unicité, mais en général il n'y a pas unicité des solutions atteignant cette valeur, on essayera de ne pas parler abusivement de **la** solution optimale.

Définition 1.6

*Le **problème de décision associé** à un problème d'optimisation Q_O est obtenu en ajoutant aux entrées de Q une valeur de seuil et en demandant s'il est possible, ou non, de trouver une solution dont la valeur est meilleure que le seuil.*

Problème d'optimisation Q_O

- ***Entrée :*** $e \in \mathcal{E}_Q$
- ***Sortie :*** $\text{opt}_{s \in sol(e)} c(s)$

Problème de décision associé Q

- ***Entrée :*** $e \in \mathcal{E}_Q$, $K \in \mathbb{N}$
- ***Sortie :*** *Existe-t-il* $s \in sol(e), c(s) \bowtie$ [a] K *?*

Exemple 1.7

Dans le cas du problème du plus court chemin dans un graphe connexe, le problème de décision associé au problème d'optimisation présenté plus haut est le problème suivant.

- **Entrée :** Un graphe orienté pondéré $G=(S,A,c)$, deux sommets s et t de S, un seuil $K \in \mathbb{N}$
- **Sortie :** Existe-t-il un chemin de longueur $\leqslant K$?

Exemple 1.8

Reprenons le problème de décision Knapsack.

Knapsack :
- **Entrée :** $n \in \mathbb{N}$, $(w_1, w_2, \ldots, w_n) \in (\mathbb{N}^\star)^n$, $(v_1, v_2, \ldots, v_n) \in (\mathbb{N}^\star)^n$, $W \in \mathbb{N}$, et $K \in \mathbb{N}$.
- **Sortie :** Existe-t-il $I \subseteq [\![1, n]\!]$ tel que $\sum_{i \in I} w_i \leqslant W$ et $\sum_{i \in I} v_i \geqslant K$?

Ce problème était en fait le problème de décision associé au problème d'optimisation suivant.

Knapsack$_o$:
- **Entrée :** $n \in \mathbb{N}$, $(w_1, w_2, \ldots, w_n) \in (\mathbb{N}^\star)^n$, $(v_1, v_2, \ldots, v_n) \in (\mathbb{N}^\star)^n$, $W \in \mathbb{N}$.
- **Sortie :** $\max \{\sum_{i \in I} v_i \mid I \subseteq [\![1, N]\!], \sum_{i \in I} w_i \leqslant W\}$

On dira alors de l'inégalité $\sum_{i \in I} w_i \leqslant W$ que c'est une **contrainte**.

Remarque 1.9

Si le problème de décision associé à un problème d'optimisation est NP difficile (comme c'est le cas pour le problème Knapsack), il est clair qu'il n'est alors pas possible (sous l'hypothèse P $\neq$ NP) d'obtenir un algorithme de complexité polynomiale pour résoudre le problème d'optimisation.
En effet la complexité du problème d'optimisation est au moins celle du problème de décision puisqu'étant muni d'un algorithme $\mathcal{O} \in \mathcal{E}_Q \to \mathbb{R}$ pour le problème d'optimisation, il est possible de fabriquer un algorithme pour le problème de décision associé de la manière suivante.

$$\mathcal{D}(e, K) : \textbf{Retourner } \mathcal{O}(e) \overset{?}{\bowtie} K$$

En effet il existe une solution dépassant le seuil K si et seulement si la valeur optimale dépasse le seuil K.

1.2 Définitions

Les algorithmes d'approximation sont des algorithmes efficaces (de complexité pire cas au plus polynomiale, souvent linéaire ou quadratique) pour fournir une solution à un problème d'optimisation,

♣. avec $\bowtie = \geqslant$ si $\text{opt} = \max$ et $\bowtie = \leqslant$ sinon.

pas nécessairement optimale, mais dont la qualité est garantie, c'est-à-dire qu'on sait a priori majorer l'écart entre la valeur fournie et l'optimum. On voit ci-dessous deux manières de mesurer cet écart à la valeur optimale.

Pour les définitions suivantes on suppose fixé un problème d'optimisation Q d'ensemble de solutions sol et de fonction objectif c. On note opt pour $\min$ s'il s'agit d'un problème de minimisation ou pour $\max$ s'il s'agit d'un problème de maximisation. On suppose de plus que la fonction objectif c est à valeurs dans $\mathbb{R}^+$

Notation 1.10

Pour $e \in \mathcal{E}_Q$, on note $\textsc{opt}(e)$ la valeur optimale pour l'instance e, i.e. $\mathrm{opt}\{c(s) \mid s \in \mathit{sol}(e)\}$.

Définition 1.11

Soit Q est un problème de maximisation. Soit $\rho \in \mathbb{R}^+$ tel que $\rho < 1$.
*On dit qu'un algorithme $\mathcal{A} \in \mathcal{E}_Q \to \mathbb{R}$ admet le **ratio d'approximation** (standard) ρ dès lors que :*

$$\forall e \in \mathcal{E}_Q,\ \mathcal{A}(e) \geqslant \rho\, \textsc{opt}(e).$$

*On dit alors que $\mathcal{A}$ est une ρ-**approximation** de Q.*

Définition 1.12

Soit Q est un problème de minimisation. Soit $\rho \in \mathbb{R}^+$ tel que $\rho > 1$.
*On dit qu'un algorithme $\mathcal{A} \in \mathcal{E}_Q \to \mathbb{R}$ admet le **ratio d'approximation** (standard) ρ dès lors que :*

$$\forall e \in \mathcal{E}_Q,\ \mathcal{A}(e) \leqslant \rho\, \textsc{opt}(e).$$

*On dit alors que $\mathcal{A}$ est une ρ-**approximation** de Q.*

Remarque 1.13

En maximisation comme en minimisation, un algorithme d'approximation est d'autant "meilleur" qu'il admet un ratio d'approximation proche de 1.

Remarque 1.14

La définition précédente est bien adaptée aux problèmes dont les solutions n'ont jamais comme valeur 0. En effet pour une instance e telle que $\textsc{opt}(e) = 0$, on demande que l'algorithme $\mathcal{A}$ fournisse lui aussi une solution de valeur 0, autrement dit qu'il soit exact. D'où la définition suivante.

Définition 1.15

Soit Q est un problème d'optimisation. On suppose que pour toute entrée $e \in \mathcal{E}_Q$, il existe une pire solution. On note $\textsc{pire}(e)$ cette valeur, i.e. la valeur des pires solutions pour l'instance e.
*On dit alors qu'un algorithme $\mathcal{A} \in \mathcal{E}_Q \to \mathbb{R}$ admet le **ratio d'approximation différentiel** $\rho < 1$ dès lors que :*

$$\forall e \in \mathcal{E}_Q,\ |\textsc{pire}(e) - \mathcal{A}(e)| \geqslant \rho\, |\textsc{pire}(e) - \textsc{opt}(e)|.$$

Remarque 1.16

Attention : que ce soit en minimisation ou en maximisation, la solution optimale est toujours plus éloignée de la pire solution qu'une solution quelconque, dans les deux cas on a donc des ratios plus petits que 1, contrairement au ratio standard qui est plus grand que 1 pour les problèmes de minimisation.

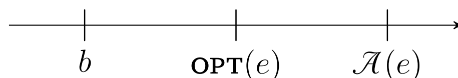
Remarque 1.17

Dans certain cas, le rapport $\frac{\mathcal{A}(e)}{\textsc{opt}(e)}$ n'est pas majoré pour $e \in \mathcal{E}_Q$, mais l'est si on se restreint aux entrées e de taille bornée par exemple, ou si l'on se restreint à un sous-ensemble d'entrées fixées par un paramètre (le degré maximal d'un graphe par exemple) .... Il y a donc des algorithmes d'approximations qui n'admettent pas de ratio d'approximation, mais qui peuvent être néanmoins intéressants, ou du moins étudiés.
On s'intéresse aussi parfois à des familles algorithmes paramétrés par la précision. Par exemple, pour chaque $\varepsilon \in \mathbb{R}_+^*$, on a un algorithme qui fournit une solution dans $[\textsc{opt}(e), (1+\varepsilon)\,\textsc{opt}(e)]$ (ou dans $[(1-\varepsilon)\,\textsc{opt}(e), \textsc{opt}(e)]$), mais sa complexité pire cas est faible si ε est considérée constante (on parle dans ce cas de **schéma d'approximation**).

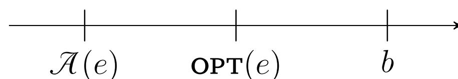
1.3 Bornes

Afin de comparer $\mathcal{A}(e)$ et $\textsc{opt}(e)$, on cherche une borne sur $\textsc{opt}(e)$.

Dans un problème de minimisation on cherche une borne inférieure b de $\textsc{opt}(e)$, à savoir un minorant de la valeur $\textsc{opt}(e)$, ainsi on a $\frac{\mathcal{A}(e)}{\textsc{opt}(e)} \leqslant \frac{\mathcal{A}(e)}{b}$. On veut le minorant le plus proche possible de $\textsc{opt}(e)$, c'est-à-dire le plus grand possible.



Dans un problème de maximisation on veut une **borne supérieure** b, c'est-à-dire un majorant de la valeur $\textsc{opt}(e)$, ainsi on a $\frac{\mathcal{A}(e)}{\textsc{opt}(e)} \leqslant \frac{\mathcal{A}(e)}{b}$. On veut le majorant le plus proche possible de $\textsc{opt}(e)$, c'est-à-dire le plus petit possible.



Plusieurs algorithmes d'approximation sont basés sur la construction d'un objet simple, qui n'est pas exactement une solution, mais qui donne à la fois une borne sur la valeur optimale et une manière de construire une solution. On voit dans les sections suivantes deux exemples de tels algorithmes d'approximation, l'un en minimisation, l'autre en maximisation. Parfois un algorithme d'approximation est simplement un algorithme glouton dont on peut justifier qu'il construit des solutions pas trop mauvaises à défaut d'être optimales.

1.4 Exemple en minimisation : Couv.Sommets

Rappel 1.18

*Une **couverture par les sommets** d'un graphe non orienté $G = (S, A)$ est un sous-ensemble de sommets $S' \subseteq S$ tel que $\forall \{u, v\} \in A, u \in S'$ ou $v \in S'$, autrement dit tel que toute arête est incidente à l'un des sommets de S'.*

Dans cette section on s'intéresse donc au problème de minimisation suivant.

$$\textsc{Couv.Sommets}_O : \begin{cases} \textbf{Entrée :} \text{ Un graphe non orienté } G = (S, A) \\ \textbf{Sortie :} \min\{\text{card}(S') \mid S' \text{ est une couverture par les sommets de } G\} \end{cases}$$

Exercice de cours 1.19

Donner la définition de Couv.Sommets le problème de décision associé à $\textsc{Couv.Sommets}_O$.

Approximation à l'aide d'un couplage. Les deux remarques suivantes expliquent comment un couplage peut-être utilisé à la fois pour donner une borne sur la valeur optimale et une solution.

- ★ Si $G = (S, A)$ admet un couplage $M \subseteq A$ ayant k arêtes, alors il faut au moins k sommets pour couvrir G. En effet, les arêtes de M étant deux à deux disjointes, un sommet ne peut couvrir qu'une seule arête de M, il faut donc au moins k sommets pour couvrir les arêtes de M, et a fortiori pour couvrir les arêtes de G.
  Un couplage fournit donc un minorant de la valeur optimale, et pour que ce minorant soit le meilleur possible, c'est-à-dire le plus grand possible, on cherche un couplage de cardinal maximum, ou du moins un couplage maximal (ce qui est plus facile à calculer).
- ✱ De plus si M est un couplage maximal, les extrémités de ses arêtes forment une couverture. En effet, notons C cet ensemble, et supposons par l'absurde qu'il existe une arête $\{u, v\} \in A$ non couverte par C, autrement dit telle que $u \notin C$ et $v \notin C$. Par construction de C, cela signifie que ni u ni v ne sont l'extrémité d'une arête de M. Autrement dit, aucune arête de M n'est incidente ni à u ni à v, ainsi $M \sqcup \{\{u, v\}\}$ est encore un couplage, ce qui nie la maximalité de M. ABSURDE La couverture C formée des extrémités de M contient exactement $2k$ sommets.
  D'après le point précédent elle contient donc au pire deux fois plus de sommets que la solution optimale, et constitue donc une solution 2-approchée.

La figure 1 représente sur un axe l'organisation des valeurs entrant en jeu dans les deux points précédents.

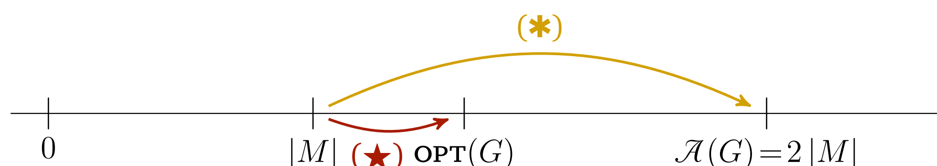
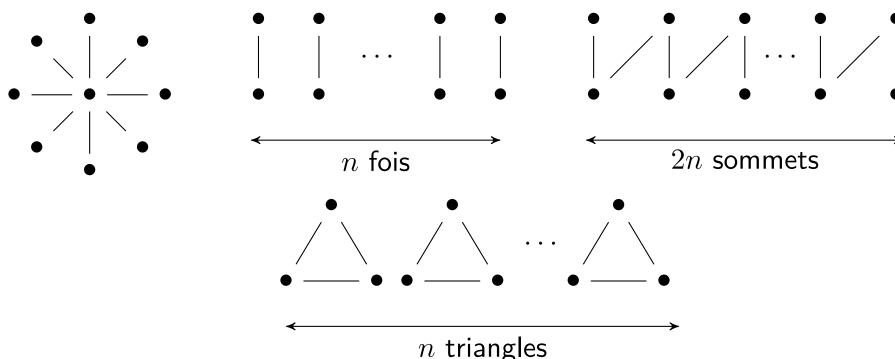


FIGURE 1 – Borne et solution pour COUV.SOMMETS$_O$ associées à un couplage M dans un graphe G

Exercice de cours 1.20

Pour les 4 graphes ci-dessous, donner un couplage maximal, la couverture associée et enfin une couverture minimum. Comparer leurs cardinaux.



Exercice de cours 1.21

Rappeler comment trouver un couplage maximal **dans le cas d'un graphe biparti**, et quelle est la complexité de cet algorithme.

Calculer un couplage maximal dans un graphe quelconque. Il suffit de partir du couplage vide, de considérer les arêtes une à une et de les ajouter si elles relient deux sommets encore libres pour

le couplage en cours de construction. Cet algorithme glouton est décrit par le pseudo-code suivant.

Algorithme 1 : CouplageGlouton (version haut niveau)

Entrée : Un graphe non orienté $G = (S, A)$
Sortie : Un couplage maximal de G

```
1 (a_i)_{i∈⟦1,m⟧} ← la liste des arêtes de G dans un ordre quelconque ;
2 C ← ∅ ;
3 i ← 1 ;
4 tant que i ⩽ m faire
5 |  {u, v} ← a_i ;
6 |  si ni u ni v ne sont l'extrémité d'une arête de C alors
7 |  |  C ← C ⊔ {{u, v}} ;
8 |  i ← i + 1 ;
9 retourner C
```

Proposition 1.22

L'algorithme CouplageGlouton est correct.

Démonstration : On rappelle qu'un couplage est maximal dès lors que lui ajouter une arête lui fait perdre sa qualité de couplage.

Soit $G = (S, A)$ un graphe non orienté. Notons $m = |A|$ et $(a_j)_{j\in[\![1,m]\!]}$ la liste de ses arêtes telle qu'elle est calculée au début de l'appel CouplageGlouton(G). Afin de montrer que cet appel renvoie bien un couplage maximal, on s'appuie sur l'invariant de boucle suivant.

$$\mathcal{I} : C \text{ est un couplage et } \forall j \in [\![1, i[\![, a_j \in C \text{ ou } C \sqcup \{a_j\} \text{ n'est pas un couplage}$$

Montrons que $\mathcal{I}$ est bien un invariant.

- D'après le lignes 2 et 3, initialement $C = \emptyset$ et $i = 1$, or $\emptyset$ est bien un couplage et $[\![1, i[\![= \emptyset$ donc $\mathcal{I}$ est vrai avant la boucle.
- Supposons que $\mathcal{I}$ est vérifié au début d'un tour de boucle quelconque, montrons qu'il l'est encore à la fin du tour. Notons C^{av} et i^{av} les valeurs des variables C et i au début du tour, et C^{ap} et i^{ap} leurs valeurs à la fin du tour. Remarquons que d'après la ligne 8, $i^{ap} = i^{av} + 1$. Par condition de boucle $i^{av} \leqslant m$, ainsi l'arête $a_{i^{av}}$ est considérée lors de ce tour. Notons-la $a = \{u, v\}$ afin d'alléger les notations.
  - Dans le cas où a n'a aucune extrémité en commun avec les arêtes de C^{av}, on entre dans la branche **alors** du **si** et $C^{ap} = C^{av} \sqcup \{a\}$. Ainsi pour $j = i^{av}$ on a bien $a_j = a \in C^{ap}$.
    Soit $j \in [\![1, i^{av}[\![$. Puisque $\mathcal{I}$ est vrai au début du tour, ou bien $a_j \in C^{av}$, et dans ce cas $a_j \in C^{ap}$ puisque $C^{av} \subseteq C^{ap}$, ou bien $C^{av} \sqcup \{a_j\}$ n'est plus un couplage, ce qui signifie que a_j intersecte l'une des arêtes de C^{av} et donc a fortiori l'une de celles de C^{ap}, donc $C^{ap} \sqcup \{a_j\}$ n'est pas un couplage non plus.
  - Dans le cas contraire, $C^{ap} = C^{av}$, donc $\mathcal{I}$ en début de tour donne $\forall j \in [\![1, i^{av}[\![, a_j \in C^{ap}$ ou $C^{ap} \sqcup \{a_j\}$ n'est pas un couplage. De plus dans ce cas là, a a une extrémité en commun avec C^{av}, soit avec C^{ap}, donc pour $j = i^{av}$ on peut affirmer que $C^{ap} \sqcup \{a_j\}$ n'est pas un couplage.

  Ainsi dans les deux cas $\mathcal{I}$ est vrai en fin de tour.

On en déduit que $\mathcal{I}$ est bien un invariant. On remarque de plus que $m + 1 - i$ est un variant de la boucle **tant que** ainsi celle-ci termine. Ainsi $\mathcal{I}$ est vrai en sortie de boucle (*i.e.* ligne 7), or en sortie de boucle i vaut $m + 1$ (en effet, on peut montrer par invariant que $i \leqslant m + 1$ et par négation de la condition de boucle que $i > m$), donc en notant C^f la valeur de C à la fin des itérations : C^f est un couplage et $\forall j \in [\![1, m]\!], a_j \in C^f$ ou $C^f \sqcup \{a_j\}$ n'est pas un couplage, et puisque $A = \{a_j \mid j \in [\![1, m]\!]\}$, cela nous donne $\forall a \in A, a \in C^f$ ou $C^f \sqcup \{a\}$ n'est pas un couplage, ou encore $\forall a \in A \setminus C^f, C^f \sqcup \{a\}$ n'est pas un couplage. Ainsi le couplage C^f retourné par CouplageGlouton(G) est bien maximal. □

Complexité Afin d'établir la complexité de cet algorithme, on précise le pseudo-code précédent.

Algorithme 2 : CouplageGlouton (version plus précise)

Entrée : Un graphe non orienté $G = (S, A)$ avec $S = [\![0, n[\![$ représenté par listes d'adjacence
Sortie : Un couplage maximal de G, représenté par tableau

```
1 C ← tableau indexé par S = ⟦0, n⟦ initialisé à None ;
2 pour tout u ∈ S faire
3     pour tout v voisin de u dans G faire
4         si C[u] = None et C[v] = None alors
5             C[u] ← v ;
6             C[v] ← u ;
7 retourner C
```

Le test ligne 4 s'effectue en temps constant, et les deux instructions à réaliser dans l'affirmative aussi (c'est tout l'intérêt d'une telle représentation du couplage C). Les boucles imbriquées lignes 2 et 3 est alors en $O(n + m)$, et comme l'initialisation du tableau ligne 1 se fait en $O(n)$, la complexité de cet algorithme est en $O(n + m)$.

Conclusion Puisqu'il est possible de calculer en temps linéaire un couplage maximal, le problème Couv.Sommets_O admet une 2-approximation en temps linéaire.

Exercice de cours 1.23

> À l'aide de l'algorithme CouplageGlouton, donner le pseudo-code de l'algorithme de 2-approximation décrit dans cette section.

1.5 Exemple en maximisation : Stable

Rappel 1.24

> *Un **stable** d'un graphe non orienté $G = (S, A)$ est un sous-ensemble de sommets $S' \subseteq S$ tel que $\forall \{u, v\} \in A, u \notin S'$ ou $v \notin S'$, autrement dit un sous-ensemble de sommets deux à deux non reliés dans G.*

Dans cette section on s'intéresse au problème de maximisation suivant.

$$\text{Stable}_O : \begin{cases} \textbf{Entrée :} \text{ Un graphe non orienté } G = (S, A) \\ \textbf{Sortie :} \max\{\text{card}(S') \mid S' \text{ est un stable } G\} \end{cases}$$

Exercice de cours 1.25

> Donner la définition de Stable le problème de décision associé à Stable_O.

Un algorithme glouton. Une idée pour construire un stable est de commencer avec le stable vide et d'y ajouter à chaque étape un sommet qui n'est ni déjà sélectionné, ni voisin d'un sommet déjà sélectionné. En vue de faire un stable avec le plus de sommets possible, on choisit à chaque étape d'ajouter le sommet qui disqualifie le moins de sommets pour la suite, c'est-à-dire celui qui a le moins de voisins parmi les sommets encore en lice. On obtient alors l'algorithme glouton ci-dessous, dans lequel S' désigne le stable en cours de construction et S l'ensemble des sommets qu'il est encore possible de choisir, c'est-à-dire des sommets hors de S' n'ayant aucun voisin dans S'.

♡. On note $\text{Vois}_S(s)$ l'ensemble des voisins dans S d'un sommet s, et $\deg_S(s)$ leur nombres, *i.e.* $\text{Vois}_S(s) = \{v \in S \mid \{v, s\} \in A\}$ et $\deg_S(s) = |\text{Vois}_S(s)|$.

Algorithme 3 : StableGlouton (version haut niveau)

Entrée : Un graphe non orienté $G = (S_0, A)$
Sortie : Un stable de G

1 $S \leftarrow$ copie de S_0 ;
2 $S' \leftarrow \emptyset$;
3 **tant que** $S \neq \emptyset$ **faire**
4 $\quad s^\star \leftarrow$ un sommet de $\arg\min\{\deg_S(s) \mid s \in S\}^{\heartsuit}$;
5 $\quad S \leftarrow S \setminus (\{s^\star\} \cup \mathsf{Vois}_S(s^\star))$;
6 $\quad S' \leftarrow S' \sqcup \{s^\star\}$;
7 **retourner** S'

Exercice de cours 1.26

Démontrer que l'algorithme 3 admet les invariants annoncés, à savoir que S' est un stable et que $\forall s \in S, \forall s' \in S', \{s, s'\} \notin A$. On pourra admettre que $S' \subseteq S_0$ et $S \subseteq S_0$ sont des invariants.

Exercice de cours 1.27

Donner un invariant permettant de justifier que le stable calculé par l'algorithme 3 est maximal.

L'algorithme précédent est de complexité polynomiale.

Exercice de cours 1.28

En admettant que P $\neq$ NP, l'algorithme StableGlouton est-il susceptible de résoudre le problème STABLE_O ?

Validité vs. optimalité. L'invariant précédent (S' est un stable), assure que l'algorithme StableGlouton renvoie bien un stable du graphe en entrée. Parfois le stable renvoyé est maximum (*Cf.* figure 2) mais ce n'est pas toujours le cas (*Cf.* figures 3 et 4). La question qui se pose alors est de savoir si le ratio entre le cardinal du stable obtenu et celui d'un cardinal maximum est minoré par $\rho > 0$ (ce qui revient à se demander si StableGlouton admet un ratio d'approximation $\rho > 1$). L'exemple de la figure 3 nous assure que si un tel ρ existe, $\rho \leqslant \frac{3}{4}$. En effet cette figure exhibe une instance e pour laquelle le stable obtenu est de cardinal 3 et un stable de cardinal 4 existe, $\frac{\mathcal{A}(e)}{\text{OPT}(e)} = \frac{3}{4} = 0.75$.

Exercice de cours 1.29

Justifier que le cardinal d'un stable maximum pour le graphe de la figure 2 est 4.
Justifier que le cardinal d'un stable maximum pour le graphe de la figure 3 est 5.
Justifier que le cardinal d'un stable maximum pour le graphe de la figure 3 est 13.

Exercice de cours 1.30

Sur les figures 2, 3 et 4 étiqueter chaque sommet encore en lice par son degré dans le graphe restant. Vérifier alors que l'exécution proposée est une exécution possible de l'algorithme StableGlouton, c'est-à-dire que le sommet sélectionné à chaque étape est bien parmi ceux de degré minimal.

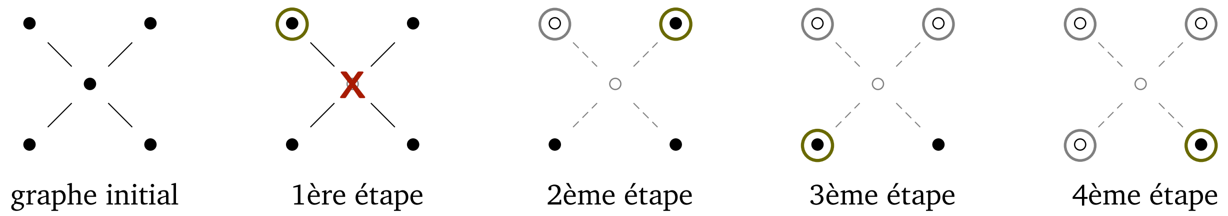


FIGURE 2 – Exécution de StableGlouton conduisant à une solution optimale

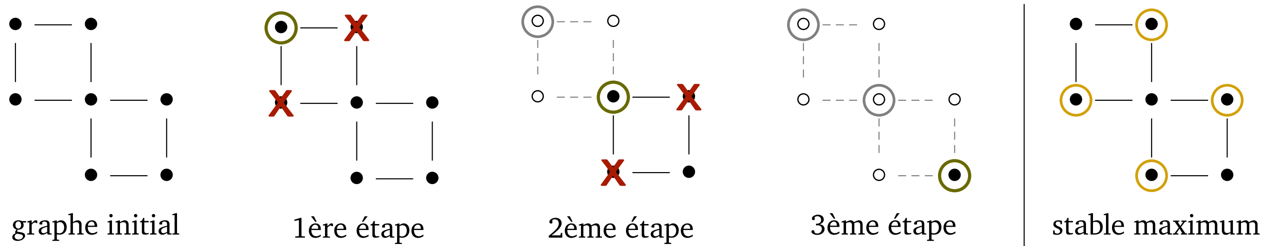


FIGURE 3 – Exécution de StableGlouton conduisant à une solution approchée (de ratio $\frac{3}{4} = 0.75$)

Contrairement à ce que pourrait laisser croire l'exemple précédent, l'algorithme StableGlouton n'admet pas un ratio d'approximation 3/4. On donne ci-dessous un exemple de graphe plus compliqué qui justifie cette affirmation.

graphe initial 1ère étape 2 étapes suivantes

4 étapes suivantes dernière étape stable maximum

FIGURE 4 – Exécution de StableGlouton conduisant à une solution approchée (de ratio $\frac{8}{13} < 0.66$)

Proposition 1.31

L'algorithme StableGlouton *admet un rapport d'approximation standard de* $\frac{1}{\Delta(G)}$ *pour les graphes de degré maximal* $\Delta(G)$ *(pour* $\Delta(G) \geqslant 1$*).*

Lemme 1.32 (*Dénombrement*)

Soit $f : A \to B$. *Si* $\forall b \in B, |f^{-1}(\{b\})| \leqslant k$, *alors* $|A| \leqslant k \times |B|$.

Démonstration : On s'appuie sur la décomposition de l'ensemble de départ de la fonction comme union des ensembles images réciproque : $A = \cup_{b\in B} f^{-1}(\{b\})$. Cette union n'étant pas nécessairement disjointe (puisque f n'est pas supposée injective), on en déduit seulement l'inégalité suivante en passant au cardinal.

$$|A| \leqslant \sum_{b\in B} \underbrace{|f^{-1}(\{b\})|}_{\leqslant k} \leqslant k|B|$$

□

Démonstration : [de la proposition 1.31] Soit $G = (S_0, A)$ un graphe non orienté de degré maximal $\Delta(G) \geqslant 1$. Soit S' le stable calculé par l'algorithme StableGlouton pour G. Soit $S^\star$ un stable maximum pour G.

Par construction, S' est maximal pour l'inclusion. En effet, à chaque étape de l'algorithme, les sommets de $S_0 \setminus S$, se séparent en deux : ceux qui ont été sélectionnés dans le stable (*i.e.* ceux de S'), et ceux qui ont été rejetés, et qui sont voisins d'un sommet du stable en construction (*i.e.* de S'). Dans un cas comme dans l'autre, ces sommets ne peuvent être ajoutés au stable. Seuls les éléments de S peuvent donc potentiellement être ajoutés à S, or quand l'algorithme s'arrête $S = \emptyset$, autrement dit aucun sommet ne peut être ajouté au stable S', ainsi S' est bien maximal.

Ainsi tout sommet de $S \setminus S'$ est voisin d'au moins un sommet de S', et c'est *a fortiori* vrai pour tout sommet de $S^\star \setminus S'$. De plus comme $S^\star$ est un stable, ce voisin d'un sommet de $S^\star$ ne peut être dans $S^\star$. Ainsi on peut associer à chaque sommet de $S^\star \setminus S'$ un sommet de $S' \setminus S^\star$ (qui lui est voisin). Comme chaque sommet de $S' \setminus S^\star$ est voisin d'au plus $\Delta(G)$ voisins, $|S' \setminus S^\star| \times \Delta(G) \geqslant |S^\star \setminus S'|$ (*Cf.* lemme 1.32). On en déduit la majoration suivante.

$$\begin{aligned}
|S^\star| &= |S^\star \cap S'| + |S^\star \setminus S'| \\
&\leqslant |S^\star \cap S'| + \Delta(G)|S' \setminus S^\star| \\
&= \Delta(G)(|S^\star \cap S'| + |S' \setminus S^\star|) \\
&= \Delta(G)|S'|
\end{aligned}$$

□

Exercice de cours 1.33

La propriété précédente ne dit rien dans le cas d'un graphe G tel que $\Delta(G) = 0$. Que dire du problème du stable maximum pour de tels graphes ? Et que renvoie l'algorithme StableGlouton pour de tels graphes ?

Remarque 1.34

La propriété 1.31, bien que parfaitement correcte, ne donne pas une bonne évaluation de l'algorithme StableGlouton. Par exemple pour des graphes de degrés maximal 2, cette propriété affirme que StableGlouton est une $\frac{1}{2}$-approximation alors que dans de tels graphes, qui se décomposent en union de chaînes, cet algorithme est optimal. Cela vient du fait que la majoration effectuée dans la preuve est grossière (mais néanmoins intéressante à savoir faire, d'où la présence de cette propriété dans le cours).

2 Séparation et évaluation

Le schéma algorithmique du “Séparation et évaluation”♣ permet de résoudre des problèmes d’optimisations NP-difficiles. On présente ce schéma dans le cas de problèmes d’optimisation linéaire en nombre entiers où il est particulièrement adapté, et on l’illustre sur une instance du problème du sac-à-dos (Knapsack).

2.1 Optimisation linéaire (continue et en nombres entiers)

2.1.1 Optimisation linéaire (continue)

L’**optimisation linéaire** est un problème d’optimisation qui consiste à maximiser ou minimiser une fonction objectif linéaire sur $\mathbb{R}^n$, sous des contraintes également linéaires♡. Quitte à changer la fonction objectif en son opposé, ce qui n’altère pas son caractère linéaire, on peut supposer qu’il s’agit d’un problème de maximisation. Le problème s’écrit alors sous la forme suivante.

$$\textsc{Ol} : \begin{cases} \textbf{Entrée :} \ (m,n) \in \mathbb{N}^2,\ A \in \mathcal{M}_{m,n}(\mathbb{R}),\ b \in \mathbb{R}^m,\ c \in \mathbb{R}^n \\ \textbf{Sortie :} \ \max\{c\cdot x \mid x \in \mathbb{R}^n, Ax \leqslant b\} \end{cases}$$

Pour une instance (n, m, A, b, c) du problème,
- n est la dimension de l’espace des solutions, les $(x_i)_{i\in[\![1,n]\!]}$ désignent n variables réelles ;
- m est le nombre d’inégalités linéaires utilisées pour décrire l’ensemble des solutions ;
- A et b définissent ces inégalités, plus précisément pour $i \in [\![1, m]\!]$, la ligne i de la matrice A, notée $A_i \in \mathbb{R}^n$, et la composante i du vecteur b définissent une inégalité, à savoir $A_i \cdot x \leqslant b_i$;
- $c \in \mathbb{R}^n$, parfois appelé direction d’optimisation, donne les coefficients de la fonction objectif.

Exemple 2.1

Calculer $\max\{4x + y \mid x \geqslant 0, y \geqslant 0, x \leqslant 2, y \leqslant 2, x + 1.6y \leqslant 9.8\}$ est un problème d’optimisation linéaire. Il s’agit en effet de l’instance (n, m, A, b, c) définie par les éléments suivants.

$$n = 2,\ m = 5,\ A = \begin{pmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1.6 \end{pmatrix},\ b = \begin{pmatrix} 0 \\ 0 \\ 2 \\ 2 \\ 9.8 \end{pmatrix}$$

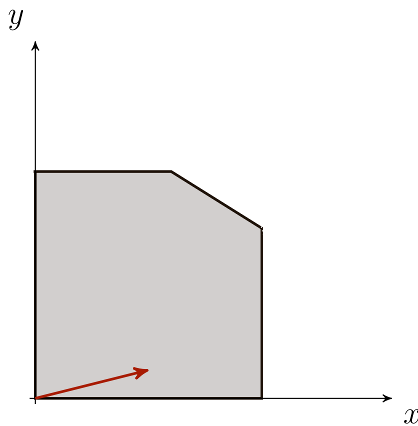
La figure 5 illustre ce qu’est un problème d’optimisation linéaire. L’instance représentée est celle donnée dans l’exemple précédent. Plus précisément :
- le polyèdre ▢ représente l’ensemble des solutions, *i.e.* l’ensemble $\{x \in \mathbb{R}^2 \mid Ax \leqslant b\}$;
- le vecteur → représente la direction d’optimisation, *i.e.* le vecteur c ;
- le point ★ représente une solution optimale (la seule ici).

Remarque 2.2

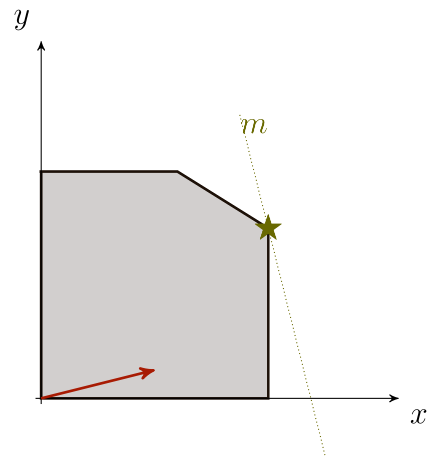
Le problème Ol est dans P, et de nombreux solveurs sont disponibles pour ce problème.

♣. *Branch-and-bound* en anglais

♡. Dans cette section on parle de fonctions et de contraintes **linéaires** comme c’est l’usage mais du point de vue mathématique c’est elles sont **affines**



(a) Une instance de OL



(b) Solution optimale de cette instance

FIGURE 5 – Illustration du problème OL

Remarque 2.3

Vocabulaire mathématique. Une égalité de la forme $A_i \cdot x = b_i$ (où $A_i \in \mathbb{R}^n$ et $b_i \in \mathbb{R}$) décrit un **hyperplan affine** (et même un hyperplan vectoriel si $b_i = 0$). Cet hyperplan sépare l'espace en deux : d'une part les points x tels que $A_i \cdot x \leqslant b_i$ et d'autre part ceux tels que $A_i \cdot x \geqslant b_i$, ces deux zones sont appelées **demi-espaces**. Une intersection finie de demi-espaces est appelée un **polyèdre**. Ainsi, puisque chacune des m inégalités d'un problème d'optimisation linéaire décrit un demi-espace, l'ensemble des solutions est un polyèdre.

Exercice de cours 2.4

On souhaite fabriquer des gâteaux très simples à base de farine, d'huile et de sucre, en proportions respectives $\frac{1}{2}$, $\frac{1}{4}$ et $\frac{1}{4}$. Ces trois produits sont vendus au kilo, à des prix respectifs de p_f€/kg, p_h€/kg et p_s€/kg. On suppose que l'on peut acheter une quantité quelconque (pas nécessairement entière) de chaque produit. On souhaite savoir combien de kilo de gâteau on peut produire au maximum avec un budget de 150€.
Donner une instance de OL qui modélise le problème.
Quelle contrainte ajouter si seulement 18kg de farine sont disponibles chez notre fournisseur ?
Quelle contrainte ajouter si le poids total de la commande chez le fournisseur ne doit pas dépasser 25kg ?

Remarque 2.5

Dans l'exercice de cours 2.4, si les produits ne sont plus vendus au poids mais par paquets de 1kg, une solution fractionnaire dans laquelle on commande 3.6kg de farine pour 1.2kg de sucre et d'huile n'est pas possible. Pour modéliser cela, il faudrait ajouter une contrainte $x \in \mathbb{Z}^3$, mais cela ne peut être modélisé par des contrainte linéaires. Plus généralement on remarque qu'une instance de OL ne permet pas de modéliser des contraintes dites d'intégrité, qui imposent aux variables de prendre des valeurs entières. Ceci motive la section suivante.

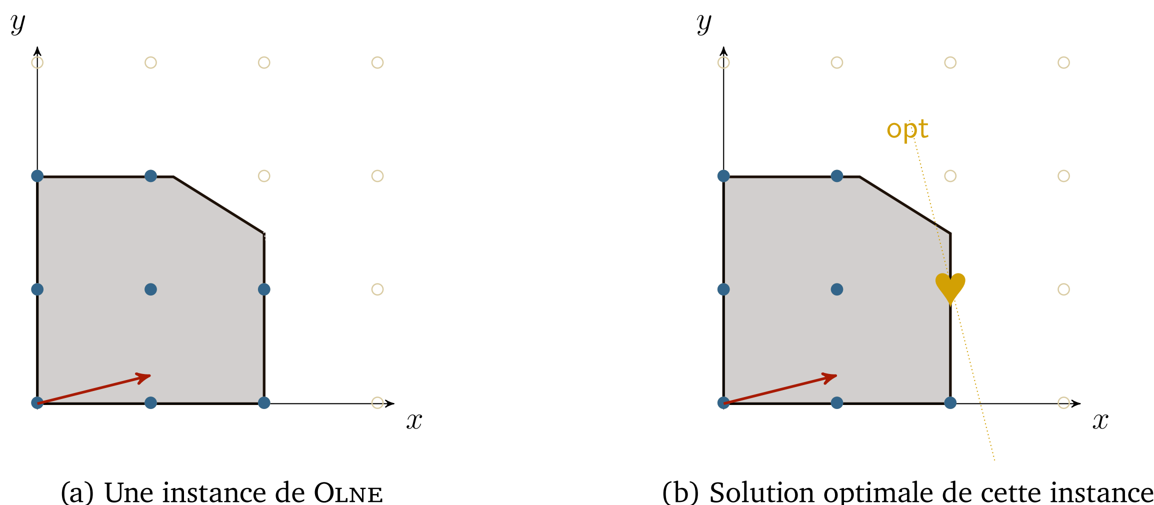
2.1.2 Optimisation linéaire en nombres entiers

Dans le cas de l'**optimisation linéaire en nombres entiers** (OLNE), les variables décisionnelles $(x_i)_{i \in [\![1,n]\!]}$ doivent prendre des valeurs entières. Le problème s'écrit donc sous la forme suivante.

$$\text{OLNE} : \begin{cases} \textbf{Entrée :} \ (m,n) \in \mathbb{N}^2, \ A \in \mathcal{M}_{m,n}(\mathbb{R}), \ b \in \mathbb{R}^m, \ c \in \mathbb{R}^n \\ \textbf{Sortie :} \ \max\{c \cdot x \mid x \in \mathbb{Z}^n, \ Ax \leqslant b\} \end{cases}$$

La figure 6 illustre ce qu'est un problème d'optimisation linéaire en nombres entiers. L'instance représentée est celle obtenue en ajoutant à celle de la figure 5 les contraintes $x \in \mathbb{Z}$ et $y \in \mathbb{Z}$.
Plus précisément :

- le polyèdre ▢ représente l'ensemble des solutions, *i.e.* l'ensemble $\{x \in \mathbb{R}^2 \mid Ax \leqslant b\}$;
- le vecteur → représente la direction d'optimisation, *i.e.* le vecteur c ;
- les points • (resp. ○) représentent les points entiers qui sont (resp. ne sont pas) solution ;
- le point ♥ représente une solution optimale (la seule ici).



(a) Une instance de OLNE (b) Solution optimale de cette instance

FIGURE 6 – Illustration du problème OLNE

Exercice de cours 2.6

Montrer que le problème KNAPSACK se réduit polynomialement au problème OLNE.

Remarque 2.7

On déduit de l'exercice de cours précédent que le problème OLNE est NP-difficile.

2.1.3 Résoudre OLNE sachant résoudre OL

Les deux problèmes OL et OLNE ont exactement les mêmes entrées : une matrice A de dimensions $n \times m$, deux vecteurs $b \in \mathbb{R}^m$ et $c \in \mathbb{R}^n$. Ainsi on peut comparer, pour une même instance (A, b, c), la solution associée à cette instance en tant que problème en nombres entiers ou en nombres réels. Une solution au problème en nombre entiers est une solution au problème en nombre réels pour la même instance, ainsi la valeur d'une solution optimale en nombres réels est toujours meilleure (au sens large) qu'une solution optimale en nombres entiers.

Vocabulaire 2.8

On dit d'un problème Q que c'est un ***relâché*** *d'un problème R lorsque Q est le même problème que R mais avec un espace de solutions élargi. On obtient par exemple un problème relâché en enlevant des contraintes au problème initial. Dans un problème de maximisation la valeur optimale d'un relâché est un majorant de la valeur optimale du problème original.*

Exemple 2.9

Le problème OL est un relâché du problème OLNE : les contraintes d'intégrité du problème OLNE ont été enlevées. On dit parfois que OL est le **relâché continu** du problème OLNE.

Dans la suite on suppose connu un algorithme OL permettant la résolution du problème OL.

Pour résoudre une instance (A, b, c) de OLNE on procède de la manière suivante. Grâce à l'algorithme OL, on résout le relâché continu associé (*i.e.* on résout (A, b, c) en tant qu'instance de OL), on obtient une solution $x^\star$ en nombres **réels**.

- Si $x^\star$ est une solution en nombres entiers, on retourne cette solution.
- Sinon $x^\star$ a au moins une coordonnée $x_i^\star$ qui n'est pas un entier.
  On considère alors les deux sous-instances suivantes :
  - (A, b, c) auquel on adjoint la contrainte linéaire $x_i \geqslant \lfloor x_i^\star \rfloor + 1$;
  - (A, b, c) auquel on adjoint la contrainte linéaire $x_i \leqslant \lfloor x_i^\star \rfloor$.

  On résout récursivement ces deux sous-instances. La solution de l'instance (A, b, c) est la meilleure des solutions obtenues pour les deux sous-instances.

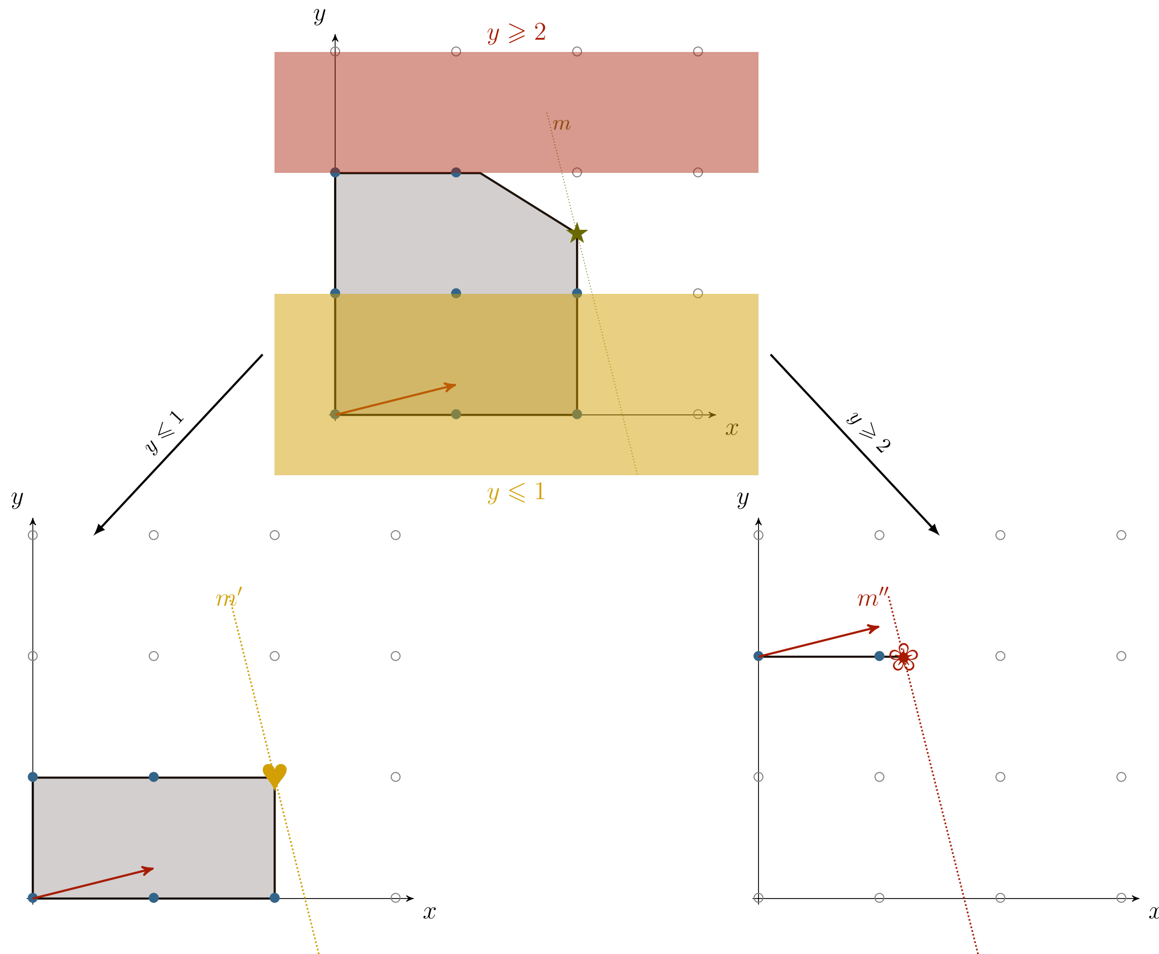


FIGURE 7 – Une étape de découpage

La figure 7 résume la situation après un découpage pour l'instance de OLNE déjà illustrée en figure 6. Comme illustré en figure 5, la solution du relâché continu ★ est $(2, 1.5)$. Cette solution n'est pas entière du fait de sa deuxième composante qui vaut 1.5, on découpe donc selon $y \leqslant 1$ ou $y \geqslant 2$.

Pour la sous-instance $y \leqslant 1$, la solution en nombres réels fournie par l'algorithme OL ♥ est en fait une solution en nombres entiers et est donc une solution optimale pour la sous-instance. Notons m' la valeur de cette solution.

Pour la sous-instance $y \geqslant 2$, la solution en nombres réels fournie par l'algorithme OL ❀ n'est pas une solution en nombre entiers, il faudrait donc a priori refaire une disjonction de cas. Toutefois, en notant m'' la valeur de ❀ , on sait que toutes les solutions en nombres entiers sont majorées

par m'', or on on remarque que $m'' < m'$. Ainsi il est inutile de poursuivre la recherche pour cette sous-instance car la solution en nombre entiers ♥ est de meilleure valeur que celle que l'on pourrait trouver pour cette sous-instance.

2.2 Le problème du sac à dos

Dans cette section on met en œuvre le paradigme du Branch-and-bound pour résoudre le problème du sac à dos, problème d'optimisation que l'on rappelle ci-dessous.

$$\text{KNAPSACK}_O : \begin{cases} \textbf{Entrée :}\ n \in \mathbb{N},\ (v_i)_{i\in[\![1,n]\!]} \in \mathbb{N}_\star^n,\ (w_i)_{i\in[\![1,n]\!]} \in \mathbb{N}_\star^n,\ W \in \mathbb{N} \\ \textbf{Sortie :}\ \arg\max\{\sum_{i=1}^n v_i x_i \mid x \in \{0,1\}^n, \sum_{i=1}^n w_i x_i \leqslant W\} \end{cases}$$

Vocabulaire 2.10

Pour une instance (n, v, w, W), on appelle ***objets*** *les éléments de $[\![0, n-1]\!]$, et pour $i \in [\![0, n-1]\!]$, v_i est appelée la* ***valeur*** *de l'objet i et w_i son* ***poids****. Enfin W est appelé la* ***capacité*** *du sac.*

Remarque 2.11

Le problème du sac à dos est un cas particulier de OLNE. En effet pour une instance (n, v, w, W) on cherche à maximiser la fonction linéaire $x \mapsto v \cdot x$ parmi les points entiers $x \in \mathbb{Z}^n$ vérifiant les contraintes linéaires suivantes : $\forall i \in [\![1, n]\!]$, $0 \leqslant \mathbb{1}_i \cdot x \leqslant 1$ et $w \cdot x \leqslant W$.

Exemple. Dans le reste de cette section, l'instance suivante du problème du sac à dos servira d'exemple. $(6, (13, 16, 19, 24, 3, 5), (6, 8, 10, 14, 2, 5), 20)$. On résume cette instance dans le tableau ci-dessous, dans lequel on fait aussi apparaître, pour chaque objet i, une valeur arrondie au dixième du rapport $\frac{v_i}{w_i}$ (qui sera utile dans la suite).

i	1	2	3	4	5	6
v_i	13	16	19	24	3	5
w_i	6	8	10	14	2	5
$\frac{v_i}{w_i}$	2.1	2.0	1.9	1.7	1.5	1.0

2.2.1 Notion de sous-instance

Une sous-instance du problème du sac à dos représente des choix déjà faits, par exemple l'objet 3 doit être présent dans la solution, l'objet 2 ne doit pas être présent dans la solution, ...Ces choix se traduisent par des contraintes comme $x_3 = 1$ ou $x_2 = 0$. Ainsi les sous-instances sont des variantes sur-contraintes de l'instance de départ, leur ensemble de solutions est donc toujours inclus dans l'ensemble de solutions de l'instance de départ.
Ces variantes sur-contraintes sont bien des instances du problème du sac à dos.

- Afin d'assurer que l'objet 3 sera présent dans une solution pour l'instance (n, v, w, W) on résout la sous-instance $(n-1, v_{|[\![1,n]\!]\setminus\{3\}}, w_{|[\![1,n]\!]\setminus\{3\}}, W - w_3)$, puis on ajoute l'objet 3 à la solution, ce qui fournit bien une solution à l'instance de départ.
- Afin d'assurer que l'objet 2 ne sera pas présent dans une solution pour l'instance (n, v, w, W) on résout la sous-instance $(n-1, v_{|[\![1,n]\!]\setminus\{2\}}, w_{|[\![1,n]\!]\setminus\{2\}}, W)$, puis on n'ajoute pas l'objet 2 à la solution, ce qui fournit bien une solution à l'instance de départ.

Remarque 2.12

Dans le cas où la sous-instance ainsi formée est non valide (si $W - w_3 \leqslant 0$ par exemple), elle correspond simplement à un sous-ensemble de solutions vide.

2.2.2 Borne des valeurs des solutions

Afin de borner les valeurs des solutions du problème du sac à dos, on s'intéresse au problème **relâché** suivant ♣.

$$\text{Knapsack}_{\mathbb{R}} : \begin{cases} \textbf{Entrée :} \; n \in \mathbb{N},\, (v_i)_{i\in[\![1,n]\!]} \in \mathbb{N}_\star^n \text{ et } (w_i)_{i\in[\![1,n]\!]} \in \mathbb{N}_\star^n,\, W \in \mathbb{N} \\ \textbf{Sortie :} \; \arg\max\{\sum_{i=1}^n v_i x_i \mid x \in [0,1]^n, \sum_{i=1}^n w_i x_i \leqslant W\} \end{cases}$$

La valeur optimale d'une instance (n, v, w, W) de Knapsack$_{\mathbb{R}}$ donne une majoration de la valeur optimale de (n, v, w, W) comme instance de Knapsack. Cette remarque relève d'un principe plus général : en maximisation, la valeur optimale d'une instance relâchée donne un majorant de la valeur optimale l'instance.

Le problème Knapsack$_{\mathbb{R}}$ admet une solution algorithmique de complexité polynomiale au moyen de l'algorithme glouton suivant.

Algorithme 4 : Résolution de Knapsack$_{\mathbb{R}}$

Entrée : Une instance (n, v, w, W) du problème Knapsack$_{\mathbb{R}}$
On suppose les suites v et w triées par ratio (v_i/w_i) décroissant.
Sortie : Une solution optimale pour l'instance (n, v, w, W) de Knapsack$_{\mathbb{R}}$

```
1 x ← vecteur de ℝ^n initialisé à 0 ;
2 R ← W ;
3 i ← 1 ;
4 tant que R > 0 et i ⩽ n faire
5 |   x_i ← min(1, R/w_i) ;
6 |   R ← R − x_i w_i ;
7 |   i ← i + 1 ;
8 retourner x ;
```

L'algorithme 4 opère de la manière suivante : on choisit en priorité les objets offrant le meilleur rapport valeur sur poids. Pour chaque objet i, on choisit de le prendre entièrement (*i.e.* $x_i = 1$) si cela est possible (tant que la capacité restante R le permet), puis lorsqu'il n'est plus possible de prendre entièrement l'objet on choisit d'en prendre la plus grande fraction possible pour la capacité restante R (*i.e.* $x_i = \frac{R}{w_i}$).

Exemple 2.13

Pour l'instance exemple, l'algorithme 4 fournit la solution en nombres réels $(1, 1, \frac{3}{5}, 0, 0, 0)$ de valeur 40.4.

Exercice de cours 2.14

Démontrer qu'à la fin de l'algorithme 4 $R + \sum_{j=1}^n x_j w_j = W$ et que $(i = n+1$ et $R \geqslant 0)$ ou $(i \leqslant n$ et $R = 0)$.

Lemme 2.15 (*admis*)

L'algorithme 4 fournit une solution optimale au problème Knapsack$_{\mathbb{R}}$.

2.2.3 Branchement

En s'inspirant de ce qui a été fait pour le problème de l'optimisation linéaire en nombres entiers on choisit dans un premier temps de "brancher" sur la variable la plus fractionnaire ♡. Cela revient

♣. La différence avec le problème Knapsack réside dans le fait que les variables décisionnelles x_i sont à valeurs dans $[0, 1]$ et non dans $\{0, 1\}$, il s'agit donc du relâché continu de Knapsack

♡. On pourrait faire un autre choix, *Cf.* exercice de cours 2.25

à diviser chaque sous-instance I en deux sous-instances en faisant une disjonction de cas sur la variable x_i, où x_i est la♣ variable de valeur fractionnaire♡ dans la solution en nombres réels que l'algorithme 4 a produit pour l'instance I. Si une telle variable x_i n'existe pas, cette solution en nombres réels est en fait une solution en nombre entiers, ainsi elle est une solution optimale pour l'instance I (et pas seulement pour son relâché). Dans ce cas, il n'est pas nécessaire de diviser l'instance I puisqu'elle est résolue.

Exemple 2.16

Puisque la solution fournie par l'algorithme 4 sur l'exemple est $(1, 1, \frac{3}{5}, 0, 0, 0)$ on choisit de brancher sur la valeur de la variable décisionnelle x_3, on considère donc les deux sous-instances suivantes : celle pour laquelle on choisit de prendre l'objet 3 ($x_3 = 1$) et celle pour laquelle on choisit de ne pas prendre l'objet 3 ($x_3 = 0$).

2.2.4 Utilisation de solutions non optimales pour élaguer

La découverte d'une solution en nombres entiers pour une sous-instance, même si elle n'est pas optimale, nous donne un minorant de la valeur optimale de l'instance de départ, et nous permet d'élaguer l'arbre de recherche. En effet, si on trouve une solution de valeur m alors qu'une sous-instance I en attente de traitement ne peut trouver que des solutions de valeurs $\leqslant M$ avec $M \leqslant m$, il est inutile de résoudre I.

Pour le problème du sac à dos, on peut trouver des solutions (non nécessairement optimales) au moyen de l'algorithme glouton suivant.

Algorithme 5 : Résolution non optimale de KNAPSACK

Entrée : Une instance (n, v, w, W) du problème KNAPSACK
On suppose les suites v et w triées par ratio (v_i/w_i) décroissant.
Sortie : Une solution en nombre entiers (non nécessairement optimale) pour (n, v, w, P)

```
1 x ← vecteur de ℤ^n initialisé à 0 ;
2 R ← W ;
3 pour i allant de 1 à n faire
4 |  si w_i ⩽ R alors
5 |  |  x_i ← 1 ;
6 |  |  R ← R − w_i
7 retourner x ;
```

L'algorithme 5 opère de la manière suivante : on choisit en priorité les objets offrant le meilleur rapport valeur sur poids.

Exemple 2.17

Pour l'instance exemple, l'algorithme 5 fournit la solution en nombres entiers $(1, 1, 0, 0, 1, 0)$ de valeur 32.

Lemme 2.18

L'algorithme 5 fournit une solution (non nécessairement optimale) au problème KNAPSACK.

Remarque 2.19

Plus la valeur trouvée pour une solution est élevée, plus elle permet d'élaguer des l'arbre de recherche.

♣. si elle existe elle est unique vu l'algorithme 4
♡. *i.e.* de valeur non entière

2.2.5 Mise en attente des sous-instances

L'ensemble des sous-instances peut être exploré de différentes manières. On mentionne ici les deux plus fréquentes :

- en profondeur (on explore en priorité la sous-instance découverte le plus récemment) ;
- ou en largeur (on explore en priorité la sous-instance en attente depuis le plus longtemps).

On pourra obtenir un parcours en profondeur au moyen d'un algorithme de Branch-and-bound récursif ou en utilisant une pile pour stocker les sous-instances en attente, on utilisera une file pour un parcours en largeur.

Remarque 2.20

> Dans le cas où l'on ne dispose pas d'algorithme fournissant une solution au problème (comme ici l'algorithme 4), ou bien que celui-ci fournit des solutions de trop mauvaise qualité (et donc de faibles minorants) l'exploration en profondeur peut être utilisée en vue de trouver une solution entière (à force de disjonction de cas, on trouve une solution entière fixée par toutes les contraintes de la branche).

Remarque 2.21

> Le parcours en largeur est motivé par l'intention d'obtenir un meilleur majorant (*Cf.* remarque 2.24) et l'espoir de trouver aussi une meilleure solution. Dans cette deuxième perspective, on peut décider de traiter à chaque étape la sous-instance la plus prometteuse (à l'image de ce que l'on fait dans l'algorithme A*), c'est-à-dire celle pour laquelle le majorant local M donné par la valeur optimale du relâché continu est le plus grand possible.

2.2.6 Mise en place d'un algorithme Branch-and-bound

On conclut cette section en présentant un algorithme de Branch-and-bound complet résolvant le problème du sac à dos.

Algorithme 6 : Algorithme de Branch-and-bound pour le problème KNAPSACK

Entrée : E une instance du problème KNAPSACK
Sortie : Une solution optimale pour E et sa valeur

```
1  meilleure_solution ← (0, 0, ..., 0) ;                //Meilleure solution trouvée
2  meilleure_valeur ← 0 ;                 //Valeur de la meilleure solution trouvée
3  todo ← {E} ;
4  tant que todo ≠ ∅ faire
5  |  Soit I une instance que l'on ôte de todo ;
6  |  Calculer s_ℝ une solution optimale du relâché de I ;  //Fournie par l'algorithme 4
7  |  M ← la valeur de la solution s_ℝ ♠ ;
8  |  si M > meilleure_valeur alors
9  |  |  Calculer s une solution de I et m sa valeur ;     //Fournie par l'algorithme 5
10 |  |  si m > meilleure_valeur alors
11 |  |  |  meilleure_valeur ← m ;
12 |  |  |  meilleure_solution ← s ;
13 |  |  si M ⩾ m + 1 alors
14 |  |  |  I_1, I_2 ← branchement(I, s_ℝ) ;
15 |  |  |  Ajouter I_1 et I_2 à todo
16 retourner (meilleure_solution, meilleure_valeur)
```

♠. On convient que $M = -\infty$ si I n'admet pas de solution

Exemple 2.22

La figure 8 présente le déroulé de l'algorithme 6 sur l'instance exemple de cette section, en mettant en avant la structure arborescente des sous-instances. Ainsi une sous-instance correspond à un nœud, et elle est définie par les choix étiquetant la branche menant à ce nœud depuis la racine.

On suppose que l'ensemble des instances à traiter todo est implémenté par une file. L'ordre dans lequel les instances sont extraites de todo est indiqué par la numération encerclée : (1), (2), ...

Lors du traitement d'une sous-instance on exécute deux algorithmes gloutons : d'abord l'algorithme 4 qui calcule une solution optimale en nombres réels (notée $\mathbb{R}$), puis l'algorithme 5 qui fournit une solution entière non nécessairement optimale ($\mathbb{Z}$). Ces solutions sont indiquées dans le nœuds de l'arbre qui représente cette sous-instance.

Sous chaque nœud, la valeur courante de meilleure_valeur au moment où la sous-instance est défilée (*i.e.* avant la calcul des solutions $\mathbb{R}$ et $\mathbb{Z}$) est indiquée dans une étoile.

À titre d'exemple le nœud de numéro (2) se lit de la manière suivante.

Lorsque la sous-instance dans laquelle $x_3 = 0$ *est considérée, la meilleure valeur connue d'une solution est* 32. *L'algorithme 5 renvoie la solution (non optimale)* $(1, 1, 0, 0, 1, 0)$ *de valeur* 32*, l'algorithme 4 renvoie la solution* $(1, 1, 0, 3/7, 0, 0)$ *de valeur* 39.2.

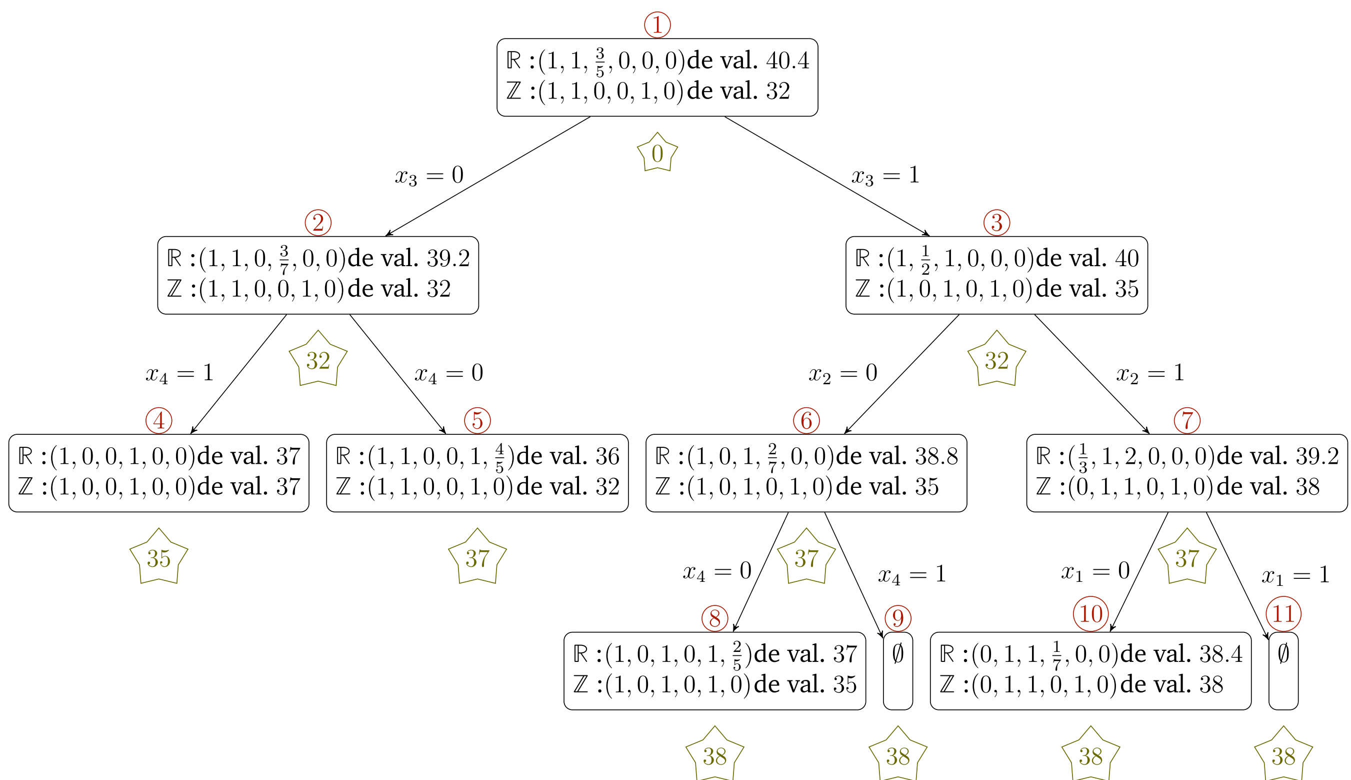


FIGURE 8 – Exécution de l'algorithme 6 sur l'instance exemple

Exemple 2.23 (*suite*)

On peut remarquer les faits suivants.

- (4). Il n'est pas utile d'explorer les sous-instances induites par (4) puisqu'on a trouvé une solution optimale en nombres réels qui est entière, ainsi c'est une solution optimale de la sous-instance (et pas seulement de son relâché). D'ailleurs on aurait pu ne pas exécuter l'algorithme 5 et ne pas calculer la solution notée $\mathbb{Z}$ qui apparaît ici sachant qu'on trouverait nécessairement une solution de valeur 37 comme celle qu'on a déjà.
- (5). Il n'est pas utile d'explorer les sous-instances induites par (5) puisqu'on ne peut espérer obtenir des solutions entières de valeur supérieure à 36 et qu'une solution de valeur 37 est déjà connue. D'ailleurs on

aurait pu ne pas exécuter l'algorithme 5 et ne pas calculer la solution notée $\mathbb{Z}$ qui apparaît ici, sachant qu'elle serait de valeur $\leqslant 36$ (de fait elle est de valeur 32.

- (8). De même, il n'est pas utile de développer ce nœud majoré par 37 alors qu'une solution de valeur 38 est déjà connue.
- (9) et (11). Les sous-instances (9) et (11) n'admettent pas de solution (car $w_3 + w_4 = 10 + 14 > 20$ et $w_3 + w_2 + w_1 = 10 + 8 + 6 > 20$).
- (10). Il n'est pas utile d'explorer les sous-instances induites par (10) puisqu'on ne peut espérer obtenir des solutions entières de valeur supérieure à 38 ($= \lfloor 38.4 \rfloor$) et qu'une solution de valeur 38 est déjà connue.

Remarque 2.24

On remarque que le majorant sur la valeur optimale dont on dispose s'améliore au cours de l'algorithme (autrement dit il décroît). Par exemple, le majorant dont on dispose après avoir résolu le relâché continu pour le nœud (1) est 40.4. Après avoir résolu les relâchés continus de ses deux fils, le majorant est $40 = \max(39.2, 40)$. En effet, ou bien la solution optimale (de l'instance de départ) vérifie $x_3 = 0$, auquel cas sa valeur est $\leqslant 39.2$ d'après le nœud (2), ou bien elle vérifie $x3 = 1$, auquel cas sa valeur est $\leqslant 40$ d'après le nœud (3), dans tous les cas elle est bien de valeur $\leqslant 40$ (et cela est plus précis que de savoir qu'elle est $\leqslant 40$[♣]).

Exercice de cours 2.25

Dans cette section on a choisi de séparer chaque sous-instance par une disjonction de cas sur la variable fractionnaire de la solution optimale du relâché continu. On peut choisir une autre stratégie de branchement, dans le cas du problème du sac à dos notamment, on peut choisir de séparer chaque sous-instance une disjonction de cas sur la variable correspondant à l'objet de plus grand poids. Par exemple sur l'instance exemple, on commence par une disjonction sur x_4 car $w_4 = 14 = \max w_i$.

Résoudre l'instance exemple par Branch-and-bound en appliquant cette stratégie de branchement.

2.3 Le principe d'un Branch-and-bound

Dans cette section on résume le principe d'un algorithme de type Branch-and-bound.

Le **Branch-and-bound** est une méthode arborescente qui permet de résoudre de manière exacte un problème d'optimisation. On peut la voir comme une amélioration de l'exploration exhaustive, ou simplement comme méthode de résolution par disjonctions de cas successives.

Séparation. La racine de l'arbre représente l'instance de départ, et chaque nœud représente une sous-instance de l'instance de départ. Afin de ne perdre aucune solution, on veillera à ce que l'union des ensembles de solutions des fils d'un nœud recouvre l'ensemble des solutions de ce nœud.
Ainsi chaque lien père-fils de l'arbre peut être étiqueté par les contraintes qui ont été ajoutées. On mentionne quelques branchements classiques :

- si x est une variable booléenne : $x = \mathsf{V}$ d'une part, $x = \mathsf{F}$ d'autre part ;
- si x est une variable numérique : $x \leqslant v$ d'une part, $x > v$ d'autre part ;
- et même si x est une variable entière : $x \leqslant v$ d'une part, $x \geqslant v + 1$ d'autre part.

Évaluation. Tout l'intérêt de cette méthode réside dans l'utilisation de bornes sur les valeurs optimales des sous-instances, afin d'élaguer l'arbre. Dans le cadre d'une maximisation, les bornes sur la valeur optimale d'une sous-instance sont de deux sortes.

♣. En réalité, sachant que la valeur opt de la solution optimale de l'instance de départ est entière, on avait dès le nœud (1) opt $\leqslant \lfloor 40.4 \rfloor$... Le propos de cette remarque reste vrai, et est aussi illustré par le développement du nœud (2).

- La valeur d'une solution quelconque de cette sous-instance est alors un minorant globale de la valeur optimale car cette solution de la sous-instance est une solution de l'instance de départ.
- La valeur d'une solution optimale du problème relâché de cette sous-instance est alors un majorant local, c'est-à-dire un majorant de cette sous-instance.

Remarque 2.26

Dans certains problèmes d'optimisation linéaire en nombres entiers, on utilise la solution optimale du relâché continu ♣pour construire une solution en nombre entiers par arrondi de cette solution fractionnaire. Cela peut être utile si on n'a pas d'algorithme efficace pour calculer une solution.

Pour le problème du sac à dos par exemple, arrondir à 0 la variable fractionnaire d'une solution forme une solution entière. On aurait pu utiliser cette remarque pour obtenir une borne inférieure à chaque sous-instance, mais on remarque que la valeur de la solution ainsi construite est moins intéressante que celle fournie par l'algorithme glouton. En effet l'arrondi se contente d'enlever le dernier objet mis dans le sac si celui-ci est fractionnaire tandis que l'algorithme glouton cherche si d'autres objets (de rapport plus petit) peuvent alors être ajoutés dans le sac.

Élagage. L'élagage consiste à tailler des branches superflues. À chaque sous-instance de majorant M, toutes les solutions sont de valeurs $\leqslant M$. On a donc plusieurs raisons de stopper le développement de cette sous-instance.
- Si on dispose d'une solution de valeur m, et que $M < m$ cela signifie qu'aucune des solutions de cette sous-instance ne sera meilleure que celle déjà connue de valeur m.
- Si la résolution du relâché de la sous-instance résout la sous-instance, il est alors inutile de subdiviser cette sous-instance, elle est déjà résolue. Dans les exemples de ce cours cela se produit lorsque la solution optimale du relâché continu est entière.
- Plus généralement, si M est égal à la valeur m d'une solution déjà connue.

Remarque 2.27

On a présenté ici le Branch-and-bound comme une méthode de résolution exacte, mais on peut aussi l'utiliser comme une méthode de résolution approchée. En effet, à chaque étape de l'algorithme on connaît un encadrement de la valeur optimale, et une solution. On peut alors choisir d'interrompre l'algorithme dès que la solution dont on dispose est, par exemple, à 5% au plus de la valeur optimale. Cette façon de faire est utilisée dans l'industrie lorsque les problèmes à traiter sont très gros, et leur résolution exacte trop coûteuse.

3 Algorithmes probabilistes

3.1 Introduction

3.1.1 Algorithmes déterministes et probabilistes.

Vocabulaire 3.1

Un algorithme $\mathcal{A}$ opérant sur un ensemble $\mathcal{E}$ est dit ***déterministe*** *si chaque exécution de $\mathcal{A}$ sur une entrée $e \in \mathcal{E}$ conduit exactement à la même suite d'états.*

♣. On utilise la solution elle-même et pas sa valeur comme plus haut.

Remarque 3.2

Cette définition implique, en particulier, qu'étant donnée une entrée $e \in \mathcal{E}$, l'exécution d'un algorithme déterministe sur e produit toujours le même résultat. Elle implique par ailleurs, qu'étant donnée une entrée $e \in \mathcal{E}$, le nombres d'étapes de calculs effectuées par l'algorithme est toujours le même quelle que soit l'exécution de l'algorithme sur l'entrée e.

Vocabulaire 3.3

Un algorithme $\mathcal{A}$ opérant sur un ensemble $\mathcal{E}$ est dit ***probabiliste*** *si la suite d'états obtenue par exécution de $\mathcal{A}$ sur une entrée $e \in \mathcal{E}$ est une variable aléatoire. Ainsi la suite d'états obtenue par exécution de $\mathcal{A}$ sur une entrée e ne dépend pas uniquement de e, mais aussi d'une source d'aléatoire.*

3.1.2 Un tri au moyen d'aléatoire.

Considérons l'algorithme de tri Bozosort ci-dessous.

Algorithme 7 : Bozosort

Entrée : Un tableau d'entiers $T[\![0, n-1]\!]$ de taille n
Sortie : Un tableau qui est une permutation triée de T

```
1 T' ← copie de T ;
2 tant que T' n'est pas trié faire
3 |  i ← U([[0, n − 1]]) ;      //Un entier tiré uniformément au hasard dans [[0, n − 1]]
4 |  j ← U([[0, n − 1]]) ;      //Un entier tiré uniformément au hasard dans [[0, n − 1]]
5 |_ échanger(T', i, j) ;
6 retourner T' ;
```

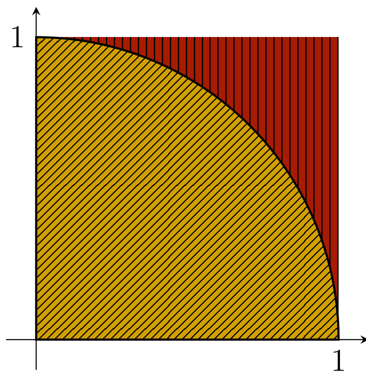
Remarque 3.4

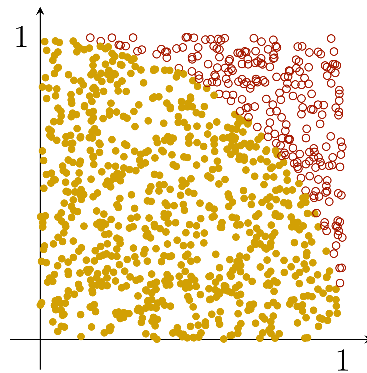
- L'exécution de cet algorithme, sur les entrées pour lesquelles il termine, conduit toujours à un tableau trié (on rappelle que l'on dit d'un tel algorithme qu'il est **partiellement correct**).
- Le temps d'exécution de l'algorithme est "plus difficile" à estimer, voire même à définir (temps moyen ? temps pire ?).
- L'algorithme peut ne pas terminer (mais on a l'intuition que ce cas est de probabilité nulle, on dit que cet algorithme termine presque sûrement[♣]).

3.1.3 Une estimation de la valeur de π au moyen d'aléatoire.

On peut faire l'observation suivante : la proportion de la surface du carré unité (*i.e.* $[0, 1] \times [0, 1]$) qui est recouverte par le disque unité (*i.e.* $\{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 \leqslant 1\}$) est de $\frac{\pi}{4}$.
Nous pouvons donc mettre en place une approximation du calcul de la surface du quart de disque en tirant uniformément des points dans le carré unité et en observant en quelle proportion ceux-ci

♣. Bien sûr, on ne dit cela qu'après l'avoir prouvé





sont dans le disque.

Algorithme 7 : Approximation de π

Entrée : Un entier $n \in \mathbb{N}^*$
Sortie : Une approximation de la valeur de π

```
1 H ← 0 ;
2 pour i = 1 à n faire
3     (x, y) ← (U[0,1], U[0,1]) ;
4     si x² + y² ⩽ 1 alors
5         H ← H + 1 ;
6 retourner 4H/n ;
```

Remarque 3.5

- Le temps d'exécution de cet algorithme ne dépend pas des choix aléatoires, il dépend uniquement de son entrée.
- À défaut de pouvoir parler de correction ici, on peut dire que la "qualité" de la réponse obtenue à l'issue d'une exécution dépend des réalisations des tirages aléatoires effectués lors de celle-ci. La qualité de la réponse obtenue dépend ici de n : plus ce paramètre est grand, meilleure sera l'approximation obtenue en moyenne sur les différentes exécutions.

Étude de l'algorithme 7. Soit $n \in \mathbb{N}^\star$, on introduit la suite de variables aléatoires $(G_i)_{i\in[\![1,n]\!]}$ indiquant si on a ajouté 1 à H au tour i, autrement dit si le tirage uniforme dans le carré $[0,1]\times[0,1]$ ligne 3 a produit un couple (x,y) tel que $x^2+y^2 \leqslant 1$. Ainsi les G_i sont des variables de Bernoulli, indépendantes et identiquement distribuées, de paramètre p où

$$p = \frac{\text{surface}(\{(x,y)\in[0,1]^2 \mid x^2+y^2\leqslant 1\})}{\text{surface}(\{(x,y)\in[0,1]^2\})} = \frac{\pi}{4}.$$

La variable aléatoire $R = \frac{4}{n}\sum_{i=1}^{n} G_i$ donne alors la valeur que retourne l'algorithme. Or par la loi faible des grands nombres :

$$\forall \varepsilon > 0, \mathbb{P}\left(\left|\frac{1}{n}\sum_{i=1}^{n} G_i - p\right| > \varepsilon\right) \xrightarrow[n\to\infty]{} 0$$

et comme $p = \frac{\pi}{4}$ on en déduit :

$$\forall \varepsilon > 0, \mathbb{P}\left(|R-\pi| > \varepsilon\right) \xrightarrow[n\to\infty]{} 0.$$

Finalement, pour chaque précision $\varepsilon \in \mathbb{R}_+^\star$, la probabilité que la valeur retournée par l'algorithme soit proche à ε près de π[♣] tend vers 1 lorsque n tend vers $+\infty$.

♣. *i.e.* qu'elle soit comprise dans l'intervalle $[\pi-\varepsilon, \pi+\varepsilon]$

3.1.4 Algorithmes de types Las Vegas et Monte-Carlo

Les exemples des sous-sections 3.1.3 et 3.1.2 motivent l'introduction des définitions suivantes♣.

Vocabulaire 3.6

> *Étant donné un problème $\mathcal{P}$, un algorithme probabiliste pour le problème $\mathcal{P}$ est dit* ***de type Las Vegas*** *dès lors que, s'il termine, c'est en donnant une réponse correcte au problème $\mathcal{P}$.*

Vocabulaire 3.7

> *Étant donné un problème $\mathcal{P}$, un algorithme probabiliste pour le problème $\mathcal{P}$ est dit* ***de type Monte-Carlo*** *dès lors que son temps d'exécution dépend uniquement de son entrée, l'algorithme pouvant cependant répondre de manière erronée au problème $\mathcal{P}$, avec une "certaine" probabilité.*

Afin d'illustrer ces deux différents types d'algorithme sur un même problème, on considère le problème suivant. Notons $\mathbb{A}_{n,p}$ l'ensemble des tableaux d'entiers de taille n contenant p occurrences de 0 et $n-p$ occurrences de 1.

$$\text{INDICE}_1 : \begin{cases} \textbf{Entrée} & : \quad T \in \mathbb{A}_{n,p} \text{ avec } n \geqslant 2 \\ \textbf{Sortie} & : \quad \text{Un indice } i \in [\![0, n[\![\text{ tel que } T[i] = 1 \end{cases}$$

Algorithme 8 : $\mathcal{A}_{\mathcal{L}}$ Algorithme de type Las Vegas résolvant INDICE$_1$

Entrée : Un tableau T de $\mathbb{A}_{n,p}$ avec $n \geqslant 2$
Sortie : Un indice $i \in [\![0, n[\![$ tel que $T[i] = 1$

```
1 i ← U[[0, n − 1]] ;
2 tant que T[i] ≠ 1 faire
3   i ← U[[0, n − 1]] ;
4 retourner i ;
```

Algorithme 9 : $\mathcal{A}_m$ Algorithme de type Monte-Carlo résolvant INDICE$_1$

Entrée : Un tableau T de $\mathbb{A}_{n,p}$ avec $n \geqslant 2$, un paramètre $k \in \mathbb{N}^\star$
Sortie : Un indice $i \in [\![0, n[\![$ tel que $T[i] = 1$ ♡

```
1 R ← −1 ;
2 pour j = 1 à k faire
3   i ← U([[0, n − 1]]) ;
4   if T[i] = 1 then
5     R ← i ;
6 retourner R ;
```

Étude de l'algorithme $\mathcal{A}_{\mathcal{L}}$. Il est clair que l'algorithme est correct lorsqu'il termine, en effet par négation de la condition de boucle $T[i] = 1$ en sortie de boucle, et la valeur de i est alors renvoyée. Fixons une entrée de l'algorithme : un tableau $T \in \mathbb{A}_{n,p}$. Notons ρ la proportion de 0 dans le tableau T, *i.e.* $\rho = \dfrac{p}{n} < 1$.

Étudions d'abord le nombre moyens d'appels au générateur de nombre aléatoire (qui est le nombre de tours de boucle plus 1) que fait l'algorithme $\mathcal{A}_{\mathcal{L}}$ sur l'entrée T. Soit $(X_l)_{l \in \mathbb{N}}$ une suite de variables

♣. On note ici que Las Vegas s'écrit sans trait d'union contrairement à Monte-Carlo.
♡. La sortie de cet algorithme est parfois -1, en pareil cas on considère que l'algorithme répond de manière éronnée, ce qui est possible pour un algorithme de type Monte-Carlo.

aléatoires indépendantes et identiquement distribuées, suivant une loi uniforme dans $[\![0, n-1]\!]$. Ces $(X_l)_{l\in\mathbb{N}}$ modélisent les résultats d'une infinité de tirages selon la loi $\mathcal{U}([\![0, n-1]\!])$, comme ceux faits dans l'algorithme. Pour $q \in \mathbb{N}^\star$, on note l'évènement A_q défini comme suit, qui correspond au cas où le rang du premier 1 de la suite $(T[X_l])$ est q.

$$A_q \overset{\text{déf}}{=} (T[X_0] = 0) \wedge (T[X_1] = 0) \wedge \ldots \wedge (T[X_{q-2}] = 0) \wedge (T[X_{q-1}] = 1).$$

Par indépendance des variables $\mathbb{P}(A_q) = \prod_{i=0}^{q} \mathbb{P}(T[X_i] = 0)$, or chaque variable X_i est uniforme, donc X_i est l'indice d'une case contenant un 0 avec probabilité ρ d'où $\mathbb{P}(A_q) = (\frac{p}{n})^{q-1}(1 - \frac{p}{n})$.
La probabilité que l'exécution de l'algorithme conduise à au plus N appels au générateur est $\mathbb{P}(\bigcup_{q=1}^{N} A_q)$, or les évènements A_q sont disjoints, donc $\mathbb{P}(\bigcup_{q=1}^{N} A_q) = \sum_{q=1}^{N} \rho^{q-1}(1-\rho) = 1 - \rho^N$.
Finalement, par théorème de continuité croissante, $\mathbb{P}(\bigcup_{n=1}^{\infty} A_q) = \lim_{N\to\infty} 1 - \rho^N = 1$. Ainsi la probabilité que l'exécution de l'algorithme s'arrête est 1, autrement dit l'algorithme termine presque sûrement.

Intéressons-nous alors à l'espérance du nombre d'itérations avant arrêt de l'algorithme. Notons pour cela Y la variable aléatoire indiquant le nombre d'itérations avant l'arrêt de l'algorithme.

$$\mathbb{E}(Y) = \sum_{q=1}^{\infty} q\, \mathbb{P}(\underbrace{Y = q}_{=A_q}) = \sum_{q=1}^{\infty} q\, \rho^{q-1}(1-\rho) = (1-\rho) \sum_{q=1}^{+\infty} q\rho^{q-1}$$

On reconnaît ici la série $\sum\limits_{n=1}^{+\infty} n\, z^{n-1}$ en $z = \rho$, qui est la dérivée de la série $\sum\limits_{n=0}^{+\infty} z^n$, qui elle-même vaut $\frac{1}{1-z}$ dès que $|z| < 1$. Ainsi en notant $f = z \mapsto \frac{1}{1-z}$, on a $f' = z \mapsto -\frac{-1}{(1-z)^2}$ et donc

$$\mathbb{E}(Y) = (1-\rho)\, f'(\rho) = (1-\rho)\, \frac{1}{(1-\rho)^2} = \frac{1}{1-\rho}$$

Autrement on peut étudier la somme partielle $S_N = \sum_{q=1}^{N} (1-\rho)\, q\, \rho^{q-1}$, où l'on a pris soin de conserver le facteur $(1-\rho)$. En développant on a alors :

$$\begin{aligned}
S_N &= \sum_{q=1}^{N} q\, \rho^{q-1} - \sum_{q=1}^{N} q\rho^q \\
&= \sum_{q=0}^{N-1} (q+1)\, \rho^q - \sum_{q=1}^{N} q\rho^q \\
&= \sum_{q=0}^{N-1} \rho^q + \Big(\sum_{q=0}^{N-1} q\rho^q - \sum_{q=1}^{N} q\rho^q \Big) \\
&= \underbrace{\sum_{q=0}^{N-1} \rho^q}_{=\frac{1-\rho^q}{(1-\rho)}} + \underbrace{\Big(0 - N\rho^N\Big)}_{\text{tend vers 0 par croiss. comp.}}
\end{aligned}$$

Donc S_N tend vers $\frac{1}{1-\rho}$ quand N tend vers $+\infty$, d'où $\mathbb{E}(Y) = \dfrac{1}{1-\rho}$

Étude de l'algorithme $\mathcal{A}_m$. L'algorithme $\mathcal{A}_m$ termine car il est seulement constitué d'une boucle **pour**. Intéressons-nous à la probabilité que cet algorithme réponde correctement au problème (pour une entrée donnée, c'est-à-dire pour un tableau T fixé et un paramètre k fixé). Pour cela intéressons-nous en fait à l'évènement complémentaire, noté E comme échec, soit E l'évènement "l'algorithme

ne retourne pas l'indice d'un 1". Il faut et il suffit pour cela que l'algorithme tire k fois l'indice d'un 0. À nouveau on utilise le fait que ces tirages sont indépendants les uns des autres, ainsi :

$$\mathbb{P}(E) = \left(\mathbb{P}\left(\mathcal{U}(\llbracket 0, n\llbracket) \in T^{-1}\{0\}\right)\right)^k = \left(\frac{\mathrm{card}(T^{-1}(\{0\}))}{\mathrm{card}(\llbracket 0, n\llbracket)}\right)^k = \rho^k$$

Finalement la probabilité que l'algorithme retourne un indice correct est $1 - \rho^k$. Cela signifie en particulier que l'on peut, en augmentant le paramètre k, augmenter la probabilité que cet algorithme retourne un résultat correct (et ce au prix d'une augmentation du nombre de calculs réalisés).

3.2 Un algorithme probabiliste de type Las Vegas : le tri rapide

Dans cette section nous étudions les algorithmes probabilistes de type Las Vegas à travers un exemple : le **tri rapide**[♣] **randomisé**.

3.2.1 Présentation de l'algorithme

Le principe de l'algorithme de tri rapide le suivant.

- Si le tableau à trier contient 0 ou 1 élément, il est déjà trié.
- Sinon,
  - on choisit un indice de pivot p dans ce tableau et on partitionne[♡] le tableau T privé de l'élément $T[p]$ en deux sous-tableaux T_1 et T_2 (les éléments inférieurs ou égaux à $T[p]$, les éléments strictement supérieurs à $T[p]$) ;
  - on trie (récursivement, en utilisant le même algorithme) les deux tableaux T_1 et T_2.

♣. aussi appelé *Quicksort* en anglais

♡. Afin d'éviter de faire grossir l'espace mémoire requis à chaque appel de l'algorithme, les sous-tableaux ne doivent pas être des copies d'une portion du tableau, mais au contraire des portions du tableau initial identifiées par un indice de début (qu'on appellera d ici) et un indice de fin (qu'on appellera f ici). La portion du tableau T comprise entre les indices d inclus et f inclus est notée $T\llbracket d, f\rrbracket$, et dans le cas où $f < d$, $T\llbracket d, f\rrbracket$ désigne le tableau vide (ou une portion vide de tableau disons).

Ces remarques conduisent aux algorithmes suivants.

Algorithme 10 : Partitionner(T, d, f, p)

Entrée : Un tableau T indicé par $[\![0, n]\!]$, trois entiers d, f et p tels que $0 \leqslant d \leqslant p \leqslant f < n$
Sortie : Un indice $q \in [\![d, f]\!]$ tel que $T[q]$ vaut finalement la valeur qu'avait $T[p]$ avant l'appel
Effet : Le sous-tableau $T[\![d, f]\!]$ est permuté de sorte que $\begin{cases} \forall k \in [\![d, q[\![, \ T[k] \leqslant T[q] \\ \forall k \in]\!]q, f[\![, \ T[k] > T[q] \end{cases}$

```
1  pivot ← T[p] ;
2  échanger T[p] et T[d] ;                    // on place le pivot au début
3  a ← d + 1 ;          //invariant : a ⩾ d + 1 et ∀k ∈ ⟦d + 1, a⟦, T[k] ⩽ pivot
4  b ← f ;                    //invariant : b ⩽ f et ∀k ∈⟧b, f⟧, T[k] > pivot
5  tant que a ⩽ b faire
6  |  si T[a] ⩽ pivot alors
7  |  |  a ← a + 1
8  |  sinon
9  |  |  échanger T[a] et T[b] ;
10 |  |  b ← b − 1 ;
11 échanger T[d] et T[a − 1] ;
12 retourner a − 1
```

Algorithme 11 : Procédure Tri_rapide(T, d, f)

Entrée : Un tableau T indicé par $[\![0, n[\![$, deux entiers $d \leqslant 0$ et $f \leqslant n - 1$
Sortie : Aucune sortie, mais $T[\![d, f]\!]$ est triée par ordre croissant

```
1 si d < f alors
2 |  p ← choix_pivot(T, d, f)          // choix d'un entier entre d et f ;
3 |  q ← Partitionner(T, d, f, p) ;
4 |  Tri_rapide(T, d, q − 1) ;
5 |  Tri_rapide(T, q + 1, f)
```

Remarque 3.8

On remarque que si $q-1 \leqslant d$, l'appel récursif ligne 4 est coupé court car la condition $d < f$ n'est pas satisfaite. De même, si $q+1 \geqslant d$, l'appel récursif ligne 5 est coupé court. En fait cette condition ligne 1 assure que l'on ne partitionne un tableau (le sous-tableau $T[\![d, f]\!]$) que s'il est composé d'au moins deux éléments.

Correction. L'algorithme Tri_rapide est correct, la preuve constitue un exercice de TD.

Exercice de cours 3.9

Quelle est la complexité **pire cas** de l'algorithme Tri_rapide dans les cas suivants :

- la fonction choix_pivot(T, d, f) retourne d (on précisera la famille de tableaux pour laquelle cette complexité pire cas est atteinte) ;
- la fonction choix_pivot(T, d, f) retourne l'indice de la médiane (on supposera que le coût de calcul de la médiane d'un sous-tableau de taille m est en $\mathcal{O}(m)$).

3.2.2 Étude de complexité en moyenne du tri rapide : choix pivot aléatoire

Dans cette sous-section on étudie le comportement de l'algorithme de tri rapide lorsque la fonction choix_pivot(T, d, f) retourne un indice tiré aléatoirement selon la loi $\mathcal{U}([\![d, f]\!])$. On ne parle pas

ici de complexité pire cas, mais on étudie, pour une entrée (T, d, f) fixée le nombre moyen de comparaisons effectuées lors de l'appel $\text{Tri_rapide}(T, d, f)$ (en moyenne sur les exécutions possibles donc).

Théorème 3.10

Le nombre moyen de comparaisons de l'algorithme de tri rapide sur une portion de tableau de taille k est en $\Theta(k \ln(k))$.

Démonstration : Hypothèses de travail. L'algorithme de tri considéré n'accède aux valeurs du tableau qu'au travers de comparaisons, ainsi son comportement est entièrement déterminé par les résultats des comparaisons 2 à 2 des éléments du tableau[♣]. Pour simplifier l'étude, on suppose aussi que les éléments du tableaux sont 2 à 2 distincts. Ainsi on se restreint à étudier les tableaux qui sont des permutations,[♡](il suffit d'assimiler chaque élément à son rang parmi les éléments du tableau). Plus précisément tous les tableaux dont il est question par la suite sont supposés être des permutations.

On remarque qu'un tel tableau est trié si et seulement si c'est l'identité, autrement dit les éléments de ces tableaux sont égaux à leur rang, *i.e.* à l'indice auquel ils doivent se trouver après le tri.

Permutations et stabilités. Pour $n \in \mathbb{N}$ et $(d, f) \in [\![0, n-1]\!]^2$ avec $d \leqslant f$, on note $\mathfrak{S}_n^{d,f}$ l'ensemble des tableaux T de taille n laissant stable $[\![d, f]\!]$. Remarquons que $\mathfrak{S}_n^{0,n-1} = \mathfrak{S}_n$, ainsi lorsqu'on appelle $\text{Tri_rapide}(T, 0, n-1)$ initialement, on a un appel de la forme $\text{Tri_rapide}(T, d, f)$ avec $T \in \mathfrak{S}_n^{d,f}$. Les appels récursifs effectués par la suite vérifient aussi cette propriété par correction de Partitionner. On note aussi Id_n la permutation identité de $[\![0, n[\![$.

Fixons une entrée. Soit T un tableau de taille n (*i.e.* une permutation de $[\![0, n[\![$). Soit $(d, f) \in [\![0, n-1]\!]^2$ tels que $T \in \mathfrak{S}_n^{d,f}$. On considère alors $X_d^f[T]$ la variable aléatoire donnant le nombre de comparaisons effectuées lors de l'exécution de $\text{Tri_rapide}(T, d, f)$, ainsi on cherche à calculer $\mathbb{E}[X_d^f[T]]$.

Tableau après partition. Pour $p \in [\![d, f]\!]$ un indice de valeur pivot donné, on note $\overline{T}^p$ le tableau obtenu après l'appel $\text{Partitionner}(T, d, f, p)$. Par correction de Partitionner et puisque $T \in \mathfrak{S}_n^{d,f}$ on peut affirmer que :

- l'indice retourné par cet appel est $T[p]$;
- $\overline{T}^p[T[p]] = T[p]$;
- $\overline{T}^p \in \mathfrak{S}_n^{d,T[p]-1}$;
- $\overline{T}^p \in \mathfrak{S}_n^{T[p]+1,f}$.

Relation de récurrence sur $\mathbb{E}[X_d^f[T]]$. Cherchons à calculer le nombre moyen de comparaisons effectuées pour trier un tableau à partir des nombres de comparaisons moyens pour les sous-tableaux.

- Si $f \leqslant d$, on ne rentre pas dans le si lors de l'appel $\text{Tri_rapide}(T, d, f)$, et l'on effectue alors aucune comparaison entre éléments du tableau, soit $X_d^f[T] = 0$. Ainsi $\mathbb{E}\left[X_d^f[T]\right] = 0$.
- Si $f > d$, l'appel $\text{Tri_rapide}(T, d, f)$ entre dans le si, tire un indice de pivot $Y \sim \mathcal{U}[\![d, f]\!]$, réalise la partition pour ce pivot puis les deux appels récursifs, ainsi

$$\begin{aligned}
\mathbb{E}\left[X_d^f[T]\right] &= \sum_{p=d}^{f} \mathbb{E}\Big[\underbrace{(f-d)}_{\text{appel à Partitionner}} + \underbrace{X_d^{T[p]-1}[\overline{T}^p]}_{\text{appel réc. l.4}} + \underbrace{X_{T[p]+1}^{f}[\overline{T}^p]}_{\text{appel réc. l.5}}\Big] \times \mathbb{P}[Y = p] && \text{par disjonction de cas} \\
&= \frac{1}{\text{card}([\![d, f]\!])} \sum_{p=d}^{f} \left(\mathbb{E}[f-d] + \mathbb{E}\left[X_d^{T[p]-1}[\overline{T}^p]\right] + \mathbb{E}\left[X_{T[p]+1}^{f}[\overline{T}^p]\right]\right) && \text{car } Y \sim \mathcal{U} \text{ et } \mathbb{E} \text{ linéaire} \\
&= (f-d) + \frac{1}{f-d+1} \sum_{p=d}^{f} \left(\mathbb{E}\left[X_d^{T[p]-1}[\overline{T}^p]\right] + \mathbb{E}\left[X_{T[p]+1}^{f}[\overline{T}^p]\right]\right) && \text{car } f-d \text{ n'est pas aléatoire}
\end{aligned}$$

♣. et par les résultats des tirages aléatoires dans le cas du tri rapide randomisé

♡. On dit qu'un tableau de taille n est une **permutation** dès lors que ses éléments sont exactement les entiers de l'intervalle $[\![0, n[\![$.

Lemme d'indépendance. $\mathbb{E}\left[X_d^f[T]\right]$ ne dépend que de $f-d$, pas de l'ordre dans lequel les éléments sont dans T. Plus formellement :

$$\forall n \in \mathbb{N}, \forall (d,f) \in [\![0,n[\![^2, f-d \leqslant l, \forall T \in \mathfrak{S}_n^{d,f}, \ \mathbb{E}\left[X_d^f[T]\right] = \mathbb{E}\left[X_0^{f-d}[\mathsf{Id}_n]\right].$$

Démonstration : Montrons par récurrence sur $f-d$ que $\mathbb{E}\left[X_d^f[T]\right]$ ne dépend que de $f-d$, pas de l'ordre dans lequel les éléments sont dans T. Formellement, on montre par récurrence sur $l \in \mathbb{N}$ la propriété suivante.

$$P_l : \forall n \in \mathbb{N}, \ \forall (d,f) \in [\![0,n[\![^2, f-d \leqslant l, \ \forall T \in \mathfrak{S}_n^{d,f}, \ \mathbb{E}\left[X_d^f[T]\right] = \mathbb{E}\left[X_0^{f-d}[\mathsf{Id}_n]\right]$$

- Soit $n \in \mathbb{N}$. Soit $(d,f) \in [\![0,n[\![^2$. N'importe quel appel à tri_rapide$(\cdot, a, b)$ avec $b \leqslant a$ ne fait aucune comparaison. Donc si $f-d \leqslant 0$, alors pour $T \in \mathfrak{S}_n^{d,f}$ on a $\mathbb{E}\left[X_d^f[T]\right] = 0 = \mathbb{E}\left[X_0^{f-d}[\mathsf{Id}_n]\right]$. Ainsi P_0 est vraie.

- Soit $l \in \mathbb{N}$. Supposons P_l vraie. Montrons P_{l+1}.
  Soient $n \in \mathbb{N}$ et $(d,f) \in [\![0,n[\![^2$ tels que $f-d = l+1$ (pour $f-d \leqslant l$, P_l permet de conclure).
  Soit $T \in \mathfrak{S}_n^{d,f}$. D'après le paragraphe précédent, on a :

$$\mathbb{E}\left[X_d^f[T]\right] = (f-d) + \frac{1}{f-d+1}\sum_{p=d}^{f} \mathbb{E}\left[X_d^{T[p]-1}[\overline{T}^p]\right] + \mathbb{E}\left[X_{T[p]+1}^f[\overline{T}^p]\right]$$

Or pour tout $p \in [\![d,f]\!]$, $\overline{T}^{T[p]} \in \mathfrak{S}_n^{d,T[p]-1}$ et $(T[p]-1)-d \leqslant (f-1)-d < f-d = l+1$, soit $(T[p]-1)-d \leqslant l$, donc d'après P_l, on a $\mathbb{E}\left[X_d^{T[p]-1}[\overline{T}^{T[p]}]\right] = \mathbb{E}\left[X_0^{T[p]-1-d}[\mathsf{Id}_n]\right]$ Or pour tout $p \in [\![d,f]\!]$, en notant $q = T[p]$, on a $\overline{T}^p \in \mathfrak{S}_n^{d,q-1}$ et $(q-1)-d \leqslant (f-1)-d < f-d = l+1$, soit $(q-1)-d \leqslant l$, donc d'après P_l, on a $\mathbb{E}\left[X_d^{T[p]-1}[\overline{T}^{T[p]}]\right] = \mathbb{E}\left[X_0^{T[p]-1-d}[\mathsf{Id}_n]\right]$.

En appliquant de même P_l sur la deuxième espérance on obtient :

$$\begin{aligned}
\mathbb{E}\left[X_d^f[T]\right] &= (f-d) + \frac{1}{f-d+1}\sum_{p=d}^{f} \mathbb{E}\left[X_0^{T[p]-1-d}[\mathsf{Id}_n]\right] + \mathbb{E}\left[X_0^{f-(T[p]+1)}[\mathsf{Id}_n]\right] \\
&= (f-d) + \frac{1}{f-d+1}\sum_{q=d}^{f} \left(\mathbb{E}\left[X_0^{q-1-d}[\mathsf{Id}_n]\right] + \mathbb{E}\left[X_0^{f-(q+1)}[\mathsf{Id}_n]\right]\right) \quad \text{chgm}^{\text{t}} \text{ d'indice } q = T[p] \text{ car } T \in \mathfrak{S}_n^{d,f} \\
&= (f-d) + \frac{1}{f-d+1}\sum_{q=d}^{f} \left(\mathbb{E}\left[X_0^{q-1-d}[\mathsf{Id}_n]\right] + \mathbb{E}\left[X_{q-d+1}^{f-d}[\mathsf{Id}_n]\right]\right)
\end{aligned}$$

Cette relation peut être appliquée à $T' = \mathsf{Id}_n$, $d' = 0$ et $f' = f - d$, ainsi on obtient

$$\begin{aligned}
\mathbb{E}\left[X_0^{f-d}[\mathsf{Id}_n]\right] &= \mathbb{E}\left[X_{d'}^{f'}[T']\right] \\
&= (f'-d') + \frac{1}{f'-d'+1}\sum_{q=d'}^{f'}\left(\mathbb{E}\left[X_0^{q-1-d'}[\mathsf{Id}_n]\right] + \mathbb{E}\left[X_0^{f'-q'-1}[\mathsf{Id}_n]\right]\right) \\
&= (f-d) + \frac{1}{f-d+1}\sum_{q=d'}^{f'}\left(\mathbb{E}\left[X_0^{q-1}[\mathsf{Id}_n]\right] + \mathbb{E}\left[X_0^{f-d-q'-1}[\mathsf{Id}_n]\right]\right) \\
&= (f-d) + \frac{1}{f-d+1}\sum_{q=d'}^{f'}\left(\mathbb{E}\left[X_0^{q+d-d-1}[\mathsf{Id}_n]\right] + \mathbb{E}\left[X_0^{f-(q+d)-1}[\mathsf{Id}_n]\right]\right) \\
&= (f-d) + \frac{1}{f-d+1}\sum_{q'=0}^{f-d}\left(\mathbb{E}\left[X_0^{q'-d-1}[\mathsf{Id}_n]\right] + \mathbb{E}\left[X_{q+1}^{f-q-1}[\mathsf{Id}_n]\right]\right) && \text{chgm}^{\text{t}}\text{ d'indice } q' = q+d \\
&= \mathbb{E}\left[X_d^f[T]\right] && \text{relation obtenue ci-avant}
\end{aligned}$$

Ainsi on a bien $\mathbb{E}\left[X_d^f[T]\right] = \mathbb{E}\left[X_0^{f-d}[\mathsf{Id}_n]\right]$. Donc P_{l+1} est vérifiée.

Par récurrence on en conclut l'indépendance annoncée.

□

On peut donc parler du nombre moyen de comparaisons réalisées sur une portion de tableau de taille donnée, sans préciser de quelle portion de quel tableau il s'agit. Ainsi on définit C_l le nombre moyen de comparaisons pour une portion de tableau à $l+1$ cases♣.

$$\forall l \in \mathbb{N} \cup \{-1\},\ C_l \stackrel{\text{déf}}{=} \mathbb{E}\left[X_0^l[\mathsf{Id}_{l+1}]\right]$$

Étude asymptotique de (C_l). $C_{-1} = C_0 = 0$ car il ne faut aucune comparaison pour trier un tableau à 0 ou 1 élément. De plus l'équation de récurrence établie plus haut se réécrit comme suit.

$$\forall l \in \mathbb{N}^*,\ C_l = l + \frac{1}{l+1}\sum_{q=0}^{l} C_{q-1}C_{l-q-1}$$

$$\begin{aligned}
\text{Donc on a } \forall n \in \mathbb{N}^*,\ C_n &= n + \frac{1}{n+1}\sum_{j=0}^{n}(C_{j-1} + C_{n-j-1}) \\
&= n + \frac{1}{n+1}\left(\sum_{j=0}^{n} C_{j-1} + \sum_{j=0}^{n} C_{n-j-1}\right) \\
&= n + \frac{1}{n+1}\left(\sum_{j=-1}^{n-1} C_j + \sum_{j=-1}^{n-1} C_j\right) \\
&= n + \frac{2}{n+1}\left(\sum_{j=1}^{n-1} C_j\right) && \text{car } C_0 = C_{-1} = 0
\end{aligned}$$

$$\text{Donc on a } \forall n \in \mathbb{N}^*,\ (n+1)C_n = n(n+1) + 2\sum_{j=1}^{n-1} C_j$$

$$\text{et de même } \forall n \in \mathbb{N}\setminus\{0,1\},\ n\,C_{n-1} = (n-1)n + 2\sum_{j=1}^{n-2} C_j$$

♣. on rappelle que si $f - d = l$, alors $[\![f, d]\!]$ a $l+1$ éléments.

En sommant ces deux relations, on obtient :

$$\begin{aligned}
\forall n\in\mathbb{N}\setminus\{0,1\},\ C_n(n+1)-C_{n-1}\,n &= n(n+1)-(n-1)n\ +\ 2\left(\sum_{j=1}^{n-1}C_j-\sum_{j=1}^{n-2}C_j\right)\\
&= n(n+1-n+1)\ +\ 2\left(\sum_{j=1}^{n-1}C_j-\sum_{j=1}^{n-2}C_j\right)\\
&= 2n\ +\ 2C_{n-1}\\
C_n(n+1)-C_{n-1}(n+2) &= 2n\\
\frac{C_n}{n+2}-\frac{C_{n-1}}{n+1} &= \frac{2n}{(n+1)(n+2)}
\end{aligned}$$

$$\begin{aligned}
\text{donc } \forall n\in\mathbb{N}\setminus\{0,1\},\ \frac{C_n}{n+2} &= \frac{C_n}{n+2}-\frac{C_0}{2}\\
&= \sum_{p=1}^{n}\left(\frac{C_n}{n+2}-\frac{C_{n-1}}{n+1}\right)\\
&= \sum_{p=1}^{n}\frac{2p}{(p+1)(p+2)}
\end{aligned}$$

Or $\frac{2p}{(p+1)(p+2)}\sim\frac{2}{p}$ et $\sum_{p\geqslant 1}\frac{2}{p}$ diverge, donc par théorème de sommation des équivalents on a $\frac{C_n}{n}\sim\sum_{p=1}^{n}\frac{2}{p}$ soit $C_n\sim n\sum_{p=1}^{n}\frac{2}{p}$. Sachant de plus que la série harmonique♣ est équivalente au logarithme, on a $C_n\sim 2n\ln(n)$.

□

3.3 Algorithmes probabilistes de type Monte-Carlo

Dans cette section nous étudions les algorithmes probabilistes de type Monte-Carlo à travers un exemple de test d'égalité dégradé. Plus précisément, en notant $\mathbb{Z}_2$ le corps des nombres $\{0,1\}$ munis des opérations d'addition modulo 2 et de la multiplication usuelle, le problème est le suivant.

ÉGALITEPRODMAT : $\begin{cases}\textbf{Entrée} & : \quad A,\ B,\ C \text{ trois matrices } n\times n \text{ sur } \mathbb{Z}_2.\\ \textbf{Sortie} & : \quad \text{A-t-on } AB \stackrel{?}{=} C\,?\end{cases}$

Une première solution au problème est l'algorithme suivant : calcule la matrice AB ; on teste coefficient par coefficient l'égalité avec C. Cet algorithme a une complexité en $\mathcal{O}(n^3)$ due au calcul du produit matriciel♡, la vérification de l'égalité étant ensuite en $\mathcal{O}(n^2)$.

On envisage une approche moins coûteuse basée sur le fait que $AB=C$ si et seulement si $\forall r\in\mathbb{Z}_2^n$, $ABr=Cr$ et que l'on peut calculer ABr non pas comme AB appliqué à r, mais comme A appliqué à Br, soit comme deux produits matrice/ vecteur, ce qui coûte de l'ordre de n^2 opérations. Autrement dit on envisage de se prononcer sur l'égalité des fonctions $X\mapsto(AB)\,X$ et $X\mapsto C\,X$ en testant seulement si ces fonctions coïncident sur un certain nombre de points choisis aléatoirement. Le lemme suivant justifie cette approche car il fournit une majoration sur la probabilité que deux matrices distinctes coïncident♠ en un point.

♣. la série des $\frac{1}{p}$

♡. On pourrait améliorer cette complexité.

♠. comprendre les applications linéaires représentées par ces matrices coïncident

Lemme 3.11

Si $D \neq 0$ est une matrice de $\mathcal{M}_n(\mathbb{Z}_2)$, et si $r \sim \mathcal{U}(\mathbb{Z}_2^n)$ alors $\mathbb{P}(Dr = 0) \leqslant \frac{1}{2}$.

Démonstration : Puisque $D \neq 0$, il existe i et j dans $[\![1, n]\!]$ tels que $D_{i,j} \neq 0$, et donc nécessairement $D_{i,j} = 1$.

Si $Dr = 0$, $\sum_{k=1}^{n} D_{i,k}\, r_k = 0$, donc $\sum_{k \neq j} D_{i,k}\, r_k = -D_{i,j} r_j$, soit $r_j = \sum_{k \neq j} D_{i,k}\, r_k$ puisque $D_{i,j} = 1$ ♣.

Par contraposée, si $r_j \neq \sum_{k \neq j} D_{i,k}\, r_k$, alors $Dr \neq 0$. On déduit de cette inclusion d'évènements que

$$\mathbb{P}(r_j \neq \sum_{k \neq j} D_{i,k}\, r_k) \leqslant \mathbb{P}(Dr \neq 0)$$

et en passant aux évènements complémentaires que

$$\mathbb{P}(Dr = 0) \leqslant 1 - \mathbb{P}(r_j \neq \sum_{k \neq j} D_{i,k}\, r_k).$$

En vue de calculer $\mathbb{P}(r_j \neq \sum_{k \neq j} D_{i,k}\, r_k)$, on considère les deux évènements suivants.

- $E_0 \overset{\text{déf}}{=} (r_j = 0 \text{ et } \sum_{k \neq j} D_{i,k} r_k = 1)$
- $E_1 \overset{\text{déf}}{=} (r_j = 1 \text{ et } \sum_{k \neq j} D_{i,k} r_k = 0)$

Alors :

$$\begin{aligned}
\mathbb{P}(r_j \neq \sum_{k \neq j} D_{i,k}\, r_k) &= \mathbb{P}(E_0 \text{ ou } E_1) = \mathbb{P}(E_0) + \mathbb{P}(E_1) && \text{car } E_0 \cap E_1 = \emptyset \\
&= \underbrace{\mathbb{P}(r_j = 0)}_{\frac{1}{2}}\, \mathbb{P}\Big(\sum_{k \neq j} D_{i,k}\, r_k = 1\Big) + \underbrace{\mathbb{P}(r_j = 1)}_{\frac{1}{2}}\, \mathbb{P}\Big(\sum_{k \neq j} D_{i,k} r_k = 0\Big) && \text{indép. des } r_k \\
&= \frac{1}{2} \left(\mathbb{P}(\sum_{k \neq j} D_{i,k} r_k = 1) + \mathbb{P}(\sum_{k \neq j} D_{i,k}\, r_k = 0) \right) \\
&= \frac{1}{2}\, \mathbb{P} \left(\sum_{k \neq j} D_{i,k} r_k = 1 \text{ ou } \sum_{k \neq j} D_{i,k}\, r_k = 0 \right) && \text{car ces évènements sont disjoints} \\
&= \frac{1}{2}
\end{aligned}$$

On conclut donc que $\mathbb{P}(Dr = 0) \leqslant 1 - \frac{1}{2} = \frac{1}{2}$. □

En appliquant le lemme à la matrice $AB - C$ on peut affirmer que si $AB \neq C$ et $r \sim \mathcal{U}(\mathbb{Z}_2^n)$, alors $\mathbb{P}(ABr = Cr) \leqslant \frac{1}{2}$. Il est intéressant de noter ici qu'en choisissant un vecteur r uniformément au hasard, la probabilité d'échec d'une détection d'un mauvais produit matriciel est bornée par $\frac{1}{2}$ qui ne dépend pas de la dimension de l'espace. On définit alors l'algorithme suivant.

Algorithme 12 : Algorithme de type Monte-Carlo testant l'égalité d'un produit matriciel

Entrée : A, B, C trois matrices $n \times n$ sur $\mathbb{Z}_2$, un paramètre k
Sortie : A-t-on $AB = C$?

```
1  Res ← vrai ;
2  pour i = 1  à  k faire
3  |   r ← U(Z_2^n) ;
4  |   if ABr ≠ Cr then
5  |   |   Res ← faux ;
6  retourner Res ;
```

♣. Le signe - disparaît car les coefficients sont dans $\mathbb{Z}_2$.

- Cet algorithme réalise k tours de boucle, et le nombre d'étapes de calcul à chaque tour est fixé par n la taille des matrices. Ainsi le temps d'exécution ne dépend pas des choix aléatoires de l'algorithme, seulement de l'entrée, c'est donc bien un algorithme **de type Monte-Carlo**.
- La complexité algorithmique de cet algorithme est de $\Theta(kn^2)$, en effet à chacun des k tours on calcule ABr comme $A(Br)$ en faisant deux produits matrice/vecteur.
- Cet algorithme ne répond jamais faux à tort (on dit qu'il est à erreur unilatérale). En effet s'il retourne faux, c'est qu'il a trouvé un vecteur r tel que $ABr \neq Cr$ et donc que $AB \neq C$.
- Dans le cas où $AB = C$, l'algorithme renvoie nécessairement vrai car on ne peut trouver de vecteur r tel que $ABr \neq Cr$.
- Enfin, dans le cas où $AB \neq C$, la probabilité d'échec, *i.e.* la probabilité que l'algorithme renvoie vrai à tort, est bornée par $(\frac{1}{2})^k$ car les choix aléatoires à chaque tour de boucle sont indépendants. En effet, en notant R_i la variable aléatoire représentant le vecteur tiré au tour de boucle i :

$$\begin{aligned}
\mathbb{P}(\mathsf{vrai} \mid AB \neq C) &= \mathbb{P}\Big(\forall i \in [\![1,k]\!], AB \cdot R_i = C \cdot R_i \,\Big|\, AB \neq C\Big) \\
&= \mathbb{P}\Big(\forall i \in [\![1,k]\!], (AB - C) \cdot R_i = 0 \,\Big|\, AB - C \neq 0\Big) \\
&= \prod_{i=1}^{k} \underbrace{\mathbb{P}\Big((AB - C) \cdot R_i = 0 \,\Big|\, AB - C \neq 0\Big)}_{\leqslant (\frac{1}{2}) \text{ d'après le lemme}} \leqslant \frac{1}{2^k} \qquad \text{par indép. des } R_i
\end{aligned}$$

Aussi en choisissant $k = 100$ (encore une fois c'est indépendant de la dimension) on a un algorithme qui a une probabilité d'échec bornée par $\frac{1}{2^{100}}$, et une complexité en $\Theta(n^2)$.

Remarque 3.12

On a présenté k comme une entrée de l'algorithme, mais on peut le voir plutôt comme un paramètre qui permet de définir un algorithme $\mathcal{A}_k$. Dans ce cas, la complexité de $\mathcal{A}_k$ est bien en $\Theta(n^2)$ car k est une constante pour cet algorithme. Cependant on ne perdra pas de vue le rôle que joue le paramètre k dans la complexité de l'algorithme. En effet si l'on souhaite assurer une erreur moyenne inférieure à $\varepsilon = 2^{-6}$ à moindre coût, alors on va utiliser l'algorithme $\mathcal{A}_6$ pour les matrices de dimension 7 ou plus, mais pour les matrices de dimensions 6 ou moins, mieux vaut calculer les $n \leqslant 6$ colonnes de AB, soit les images par AB des n vecteurs de bases, que $k \geqslant 6$ images de vecteurs choisis aléatoirement... Autrement dit en assez petite dimension on fait le test d'égalité usuel.