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

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

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

Exemple 1.2
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 1.3

*Bien que dans le cadre d'un problème quelconque $Q \subseteq \mathcal{E}_Q \times \mathcal{S}_Q$ on a 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 1.4

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 1.5

*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	*Problème de décision associé Q*
$\begin{cases} \textbf{\textit{Entrée :}}\ e \in \mathcal{E}_Q \\ \textbf{\textit{Sortie :}}\ \mathrm{opt}_{s \in \mathit{sol}(e)}\, c(s) \end{cases}$	$\begin{cases} \textbf{\textit{Entrée :}}\ e \in \mathcal{E}_Q,\ K \in \mathbb{N} \\ \textbf{\textit{Sortie :}}\ \textit{Existe-t-il}\ s \in \mathit{sol}(e), c(s) \bowtie {}^{a} K\ ? \end{cases}$

Exemple 1.6
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.

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

Exemple 1.7
Reprenons le problème de décision KNAPSACK.

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

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

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

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

Remarque 1.8

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 $\mathsf{P} \neq \mathsf{NP}$) 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 $\mathcal{O} \in \mathcal{E}_{\mathsf{Q}} \to \mathbb{R}$ pour le problème d'optimisation, il est possible de fabriquer un algorithme pour le problème de décision associé de la manière suivante.

$$\mathcal{D}(e, K) = \mathcal{O}(e) \overset{?}{\bowtie} K$$

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, 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.

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

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 $\max$ 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 1.9

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 1.10

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 1.11

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 1.12

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

Remarque 1.13

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 1.14

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 1.15

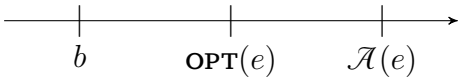
Attention : que ce soit en minimisation ou en maximisation, la solution optimale est toujours plus éloignée de la pire solution qu'une solution quelconque, dans les deux cas on a donc des ratios plus petits que 1, contrairement au ratio standard qui est plus grand que 1 pour les problèmes de minimisation.

Remarque 1.16

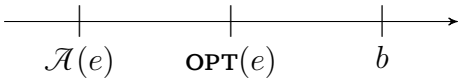
Dans certain cas, le rapport $\frac{\mathcal{A}(e)}{\text{OPT}(e)}$ n'est pas majoré pour $e \in \mathcal{E}_Q$, mais l'est si on se restreint aux entrées e de taille bornée par exemple, ou si l'on se restreint à un sous-ensemble d'entrées fixées par un paramètre (le degré maximal d'un graphe par exemple) …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 $[\text{OPT}(e), (1+\varepsilon)\,\text{OPT}(e)]$ (ou dans $[(1-\varepsilon)\,\text{OPT}(e), \text{OPT}(e)]$), mais sa complexité pire cas est faible si ε est considérée constante (on parle dans ce cas de schéma d'approximation).

1.3 Bornes

Comment comparer $\mathcal{A}(e)$ et $\text{OPT}(e)$ si on ne connaît pas $\text{OPT}(e)$ …L'idée c'est d'avoir *une borne* sur $\text{OPT}(e)$. Dans un problème de mininimisation on veut une **borne inférieure** b, c'est-à-dire un minorant de la valeur $\text{OPT}(e)$, ainsi on a $\frac{\mathcal{A}(e)}{\text{OPT}(e)} \leqslant \frac{\mathcal{A}(e)}{b}$. Et évidemment on veut le minorant le plus proche possible de $\text{OPT}(e)$, c'est-à-dire le plus grand possible. Une bonne borne inférieure est une grande borne inférieure.



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



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 on fait juste 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 1.17

*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'.*

La problème de la couverture par les sommets On remarque que l'ensemble S entier est toujours une couverture par les sommets, la question que l'on peut alors se poser est de savoir comment couvrir un graphe avec un minimum de sommets. Dans cette section on s'intéresse donc au problème de minimisation suivant.

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

Exercice de cours 1.18

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.
- ✱ De plus si M est un couplage est 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.

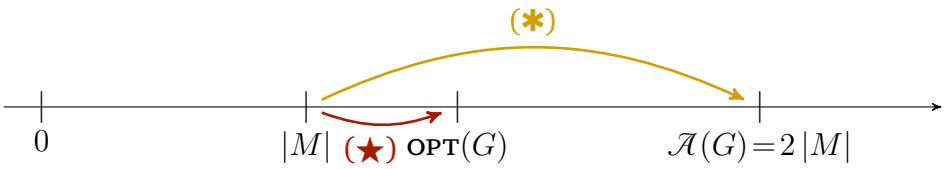
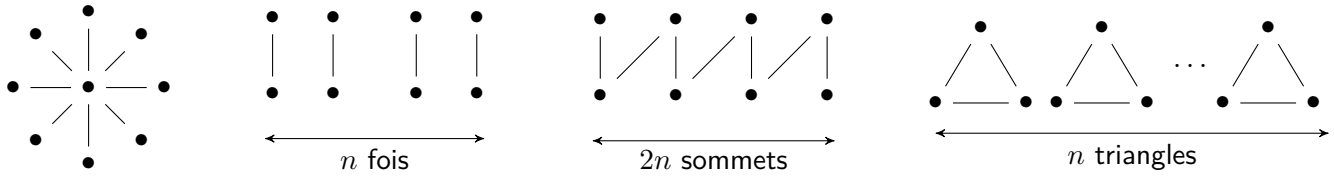


Figure 1 – Borne et solution pour $\textsc{Couv.Sommets}_O$ associées à un couplage M dans un graphe G

Exercice de cours 1.19

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



Exercice de cours 1.20

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

Comment 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 haut niveau suivant.

Algorithme 1 : CouplageGlouton (version haut niveau)

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

```
1 (a_j)_{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_j ;
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
```

Correction On peut démontrer que la propriété "C est un couplage de G" est un invariant, ainsi il suffit de démontrer la maximalité du couplage retourné. 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'appuye sur l'invariant de boucle suivant.

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

- D'après le lignes 2 et 3, initialement $C = \emptyset$ et $i = 1$, or $\emptyset$ est bien un couplage et $[\![1, i[\![= \emptyset$ donc l'invariant est vrai avant la boucle.
- Supposons que l'invariant 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, $i^{ap} = i^{av} + 1$. Notons $a = \{u, v\}$ l'arête $a_{i^{av}}$ qui est considérée lors de ce tour.
  - Dans le cas où a n'a aucune extrémité en commun avec les arêtes de C^{av}, on entre dans le si et alors $C^{ap} = C^{av} \sqcup \{a\}$. Ainsi pour $j = i^{av}$ on a bien $a_j = a \in C^{ap}$.
    Soit $j \in [\![1, i^{av}[\![$. Puisque l'invariant 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 l'invariant en début de tour donne directement $\forall j \in [\![1, i^{av}[\![, a_j \in C^{ap}$ 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 l'invariant est vrai en fin de tour.

On en déduit que l'invariant est vrai en sortie de boucle. On observe qu'en sortie de boucle i vaut m, donc en notant C^{fin} la valeur de C on a $\forall j \in [\![1, m]\!], a_j \in C^{fin}$ ou $C^{fin} \sqcup \{a_j\}$ n'est pas un couplage, et puisque $A = \{a_j \mid j \in [\![1, m]\!]\}$, cela se réécrit $\forall a \in A, a \in C^{fin}$ ou $C^{fin} \sqcup \{a\}$ n'est pas un couplage, ou encore $\forall a \in A \setminus C^{fin}, C^{fin} \sqcup \{a\}$ n'est pas un couplage. Ainsi le couplage C^{fin} 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 $S = [\![0, n[\![$ codé par listes d'adjacence
Sortie : Un couplage maximal de G, codé 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 1.21

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 1.22

*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.*

La problème du stable maximum. Dans cette section on s'intéresse au problème de minimisation suivant.

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

Exercice de cours 1.23

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'.

Algorithme 3 : StableGlouton (version haut niveau)

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

```
1 S ← copie de S_0 ;
2 S' ← ∅ ;
3 tant que S ≠ ∅ faire
4     s* ← un sommet de arg min{deg_S(s) | s ∈ S} ♡ ;
5     S ← S \ ({s*} ∪ Vois_S(s*)) ;
6     S' ← S' ⊔ {s*} ;
7 retourner S'
```

Exercice de cours 1.24

Démontrer que l'algorithme StableGlouton satisfait 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 les invariants $S' \subseteq S_0$ et $S \subseteq S_0$ sont satisfaits.

L'algorithme précédent est de complexité polynomiale.

Exercice de cours 1.25

En admettant que P $\neq$ NP, l'algorithme StableGlouton est-il susceptible de résoudre le problème $\textsc{Stable}_O$?

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, $\frac{\mathcal{A}(e)}{\textsc{opt}(e)} = \frac{3}{4} = 0.75$. L'exemple de la figure 4) exhibe une instance e' plus mauvaise : le stable obtenu est de cardinal 8 et un stable de cardinal 13, $\frac{\mathcal{A}(e')}{\textsc{opt}(e')} = \frac{8}{13} \leqslant 0.66$.

Exercice de cours 1.26

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 13.

Exercice de cours 1.27

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.

♡. 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 $deg_S(s) = |\text{Vois}_S(s)|$.

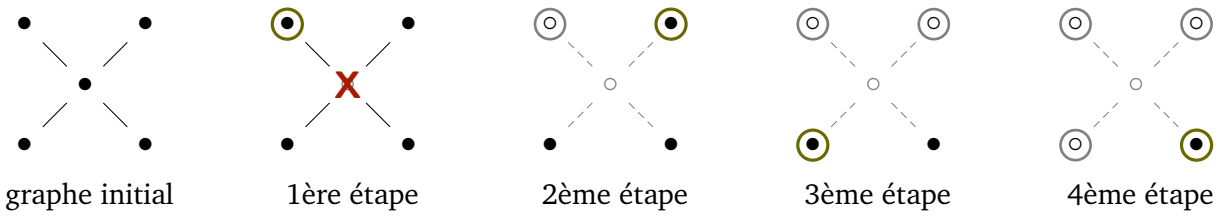


FIGURE 2 – Exécution de StableGlouton conduisant à une solution optimale

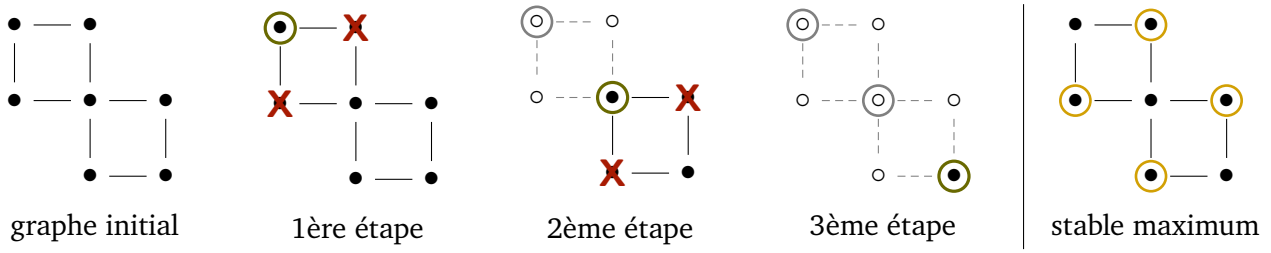
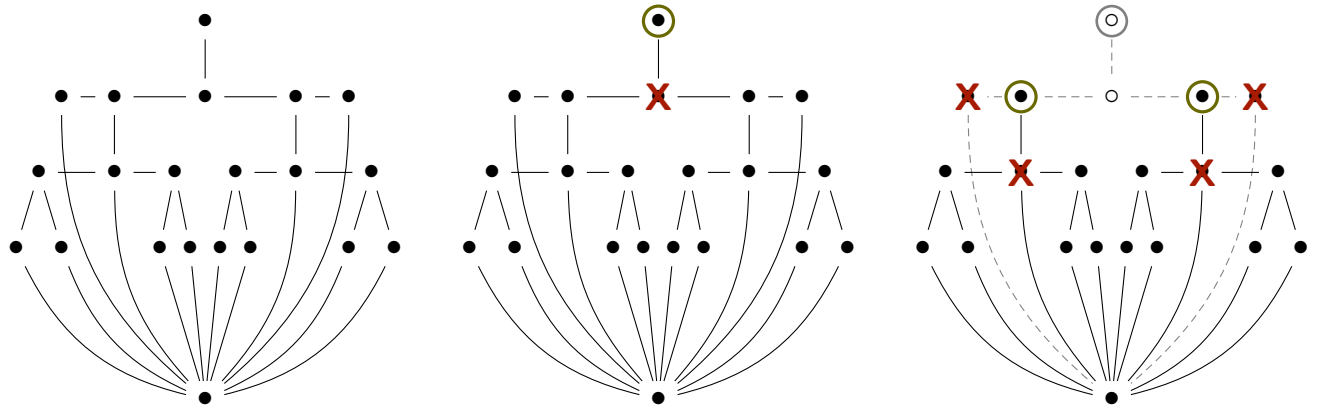
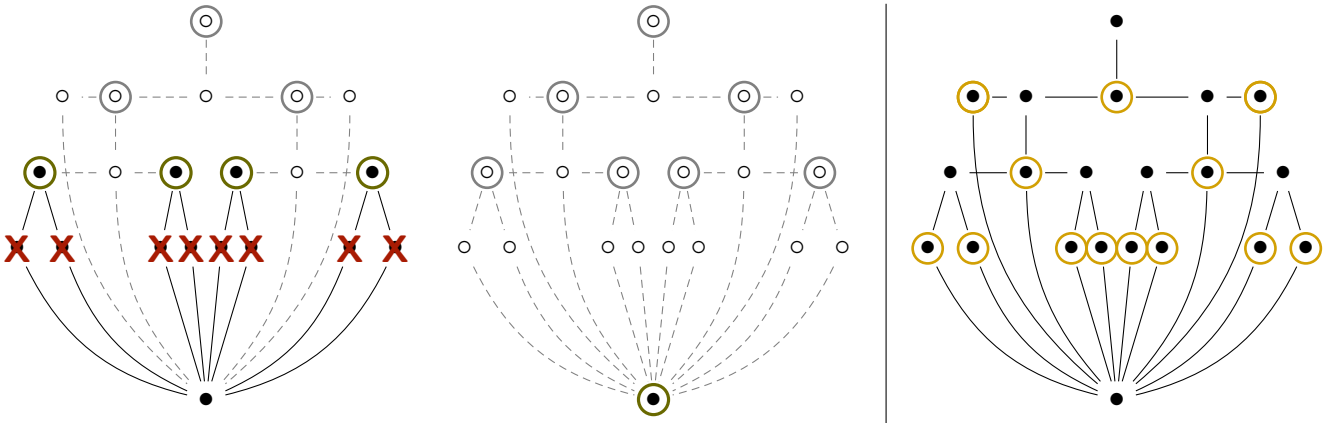


FIGURE 3 – Exécution de StableGlouton conduisant à une solution approchée (de ratio $\frac{3}{4} = 0.75$)

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 1.28

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 1.29 (*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'appuye 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 1.28 : Soit $G = (S_0, A)$ un graphe non orienté de degré max. $\Delta(G) \geqslant 1$. 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'arête $S = \emptyset$, 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 $S^* \setminus S'$. De plus comme S^* est un stable, ce voisin d'un sommet de S^* ne peut être dans S^*. Ainsi on peut associer à chaque sommet de $S^* \setminus S'$ un sommet de $S' \setminus S^*$ (qui lui est voisin). Comme chaque sommet de $S' \setminus S^*$ étant voisin d'au plus $\Delta(G)$ voisins, on a $|S' \setminus S^*| \times \Delta(G) \geqslant |S^* \setminus S'|$ (*Cf.* lemme 1.29). On en déduit la majoration suivante.

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

□

Exercice de cours 1.30

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 1.31

La propriété 1.28, 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.

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

Pour une instance (n, m, A, b, c) du problème,

♣. *Branch-and-bound* en anglais

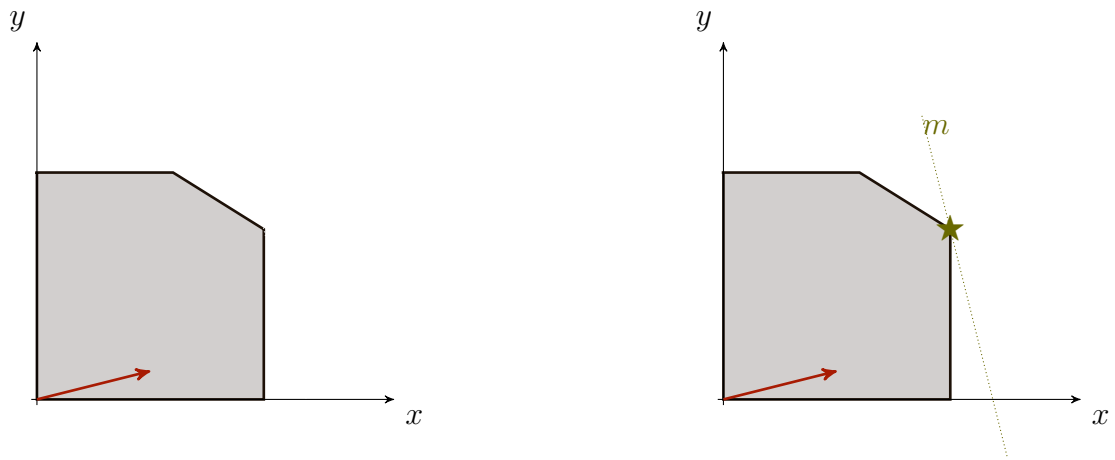
♡. Dans cette section on parle de fonctions et de contraintes **linéaires** comme c'est l'usage mais du point de vue mathématique c'est elles sont **affines**

- 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).



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

FIGURE 5 – Illustration du problème OL

Remarque 2.2

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

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

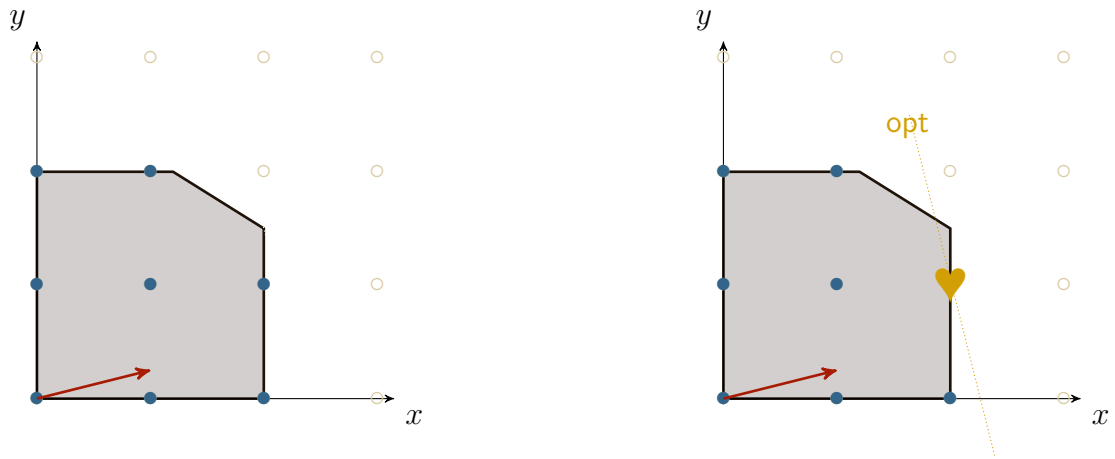
Dans l'exercice de cours 2.4, si les produits ne sont plus vendus au poids mais par paquets de 1kg, une solution fractionnaire dans laquelle on commande 3.6kg de farine pour 1.2kg de sucre et d'huile n'est pas possible. Pour modéliser cela, il faudrait ajouter une contrainte $x \in \mathbb{Z}^3$, mais 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.

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.

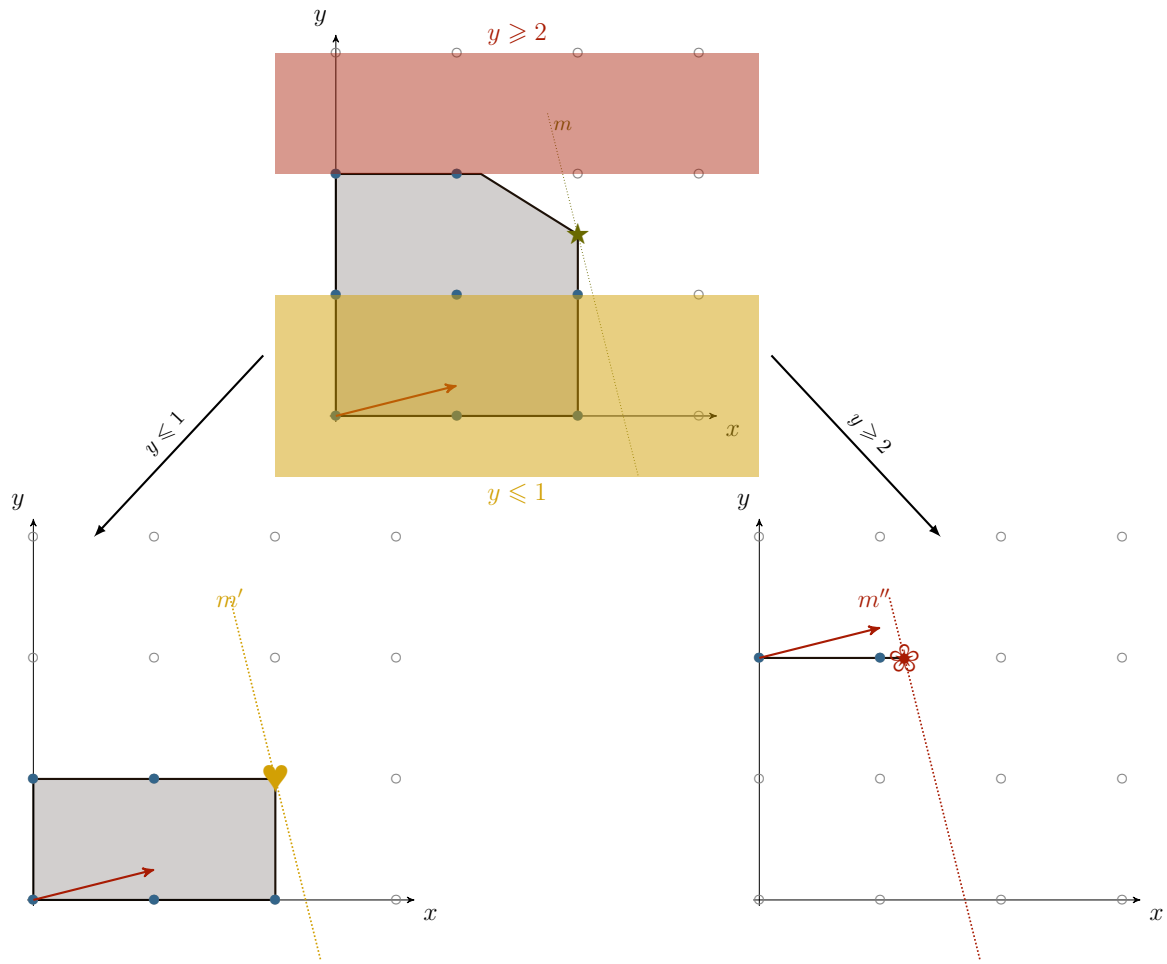


FIGURE 7 – Une étape de découpage

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.

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

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 P est appelé la **capacité** du sac.*

Le choix du problème du sac à dos est motivé par le fait que c'est un problème d'optimisation linéaire en nombre entiers, autrement dit 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, …
De tels choix peuvent être représentés en ajoutant des contraintes linéaires.

- Afin d'assurer que l'objet 3 sera présent dans la solution, on force la variable x_3 à prendre la valeur 1 en ajoutant l'inégalité linéaire $\mathbb{1}_3 \cdot x \geqslant 1$ (on a déjà l'inégalité $\mathbb{1}_3 \cdot x \leqslant 1$ dans l'instance de départ).
- Afin d'assurer que l'objet 2 ne sera pas présent dans la solution, on force la variable x_2 à prendre la valeur 0 en ajoutant l'inégalité linéaire $\mathbb{1}_2 \cdot x \leqslant 0$ (on a déjà l'inégalité $\mathbb{1}_2 \cdot x \geqslant 0$ dans l'instance de départ).

Ansi 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.

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♣.

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

La valeur optimale d'une instance (n, v, w, W) de $\textsc{Knapsack}_{\mathbb{R}}$ 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 $\textsc{Knapsack}_{\mathbb{R}}$ admet une solution algorithmique de complexité polynomiale au moyen

♣. 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

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$_{\mathbb{R}}$

```
x ← vecteur de ℝ^n initialisé à 0 ;
R ← W ;
i ← 1 ;
tant que R > 0 et i ⩽ n faire
    x_i ← min(1, R/w_i) ;
    R ← R − x_i w_i ;
    i ← i + 1 ;
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 2.11
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 2.12

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 2.13 (*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 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 2.14
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 ($x_3 = 0$).

♣. On pourrait faire un autre choix, *Cf.* exercice de cours 2.22
♡. si elle existe elle est unique vu l'algorithme 4
♠. *i.e.* de valeur non entière

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 2.15
Pour l'instance exemple, l'algorithme 5 fournit la solution en nombres entiers $(1, 1, 0, 0, 1, 0)$ de valeur 32.

Lemme 2.16

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

Remarque 2.17

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

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 2.18

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 patr toutes les contraintes de la branche).

Remarque 2.19

Le parcours en largeur est motivé par l'intention d'obtenir un meilleur majorant (*Cf.* remarque 2.21) 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)
```

Exemple 2.20

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.

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

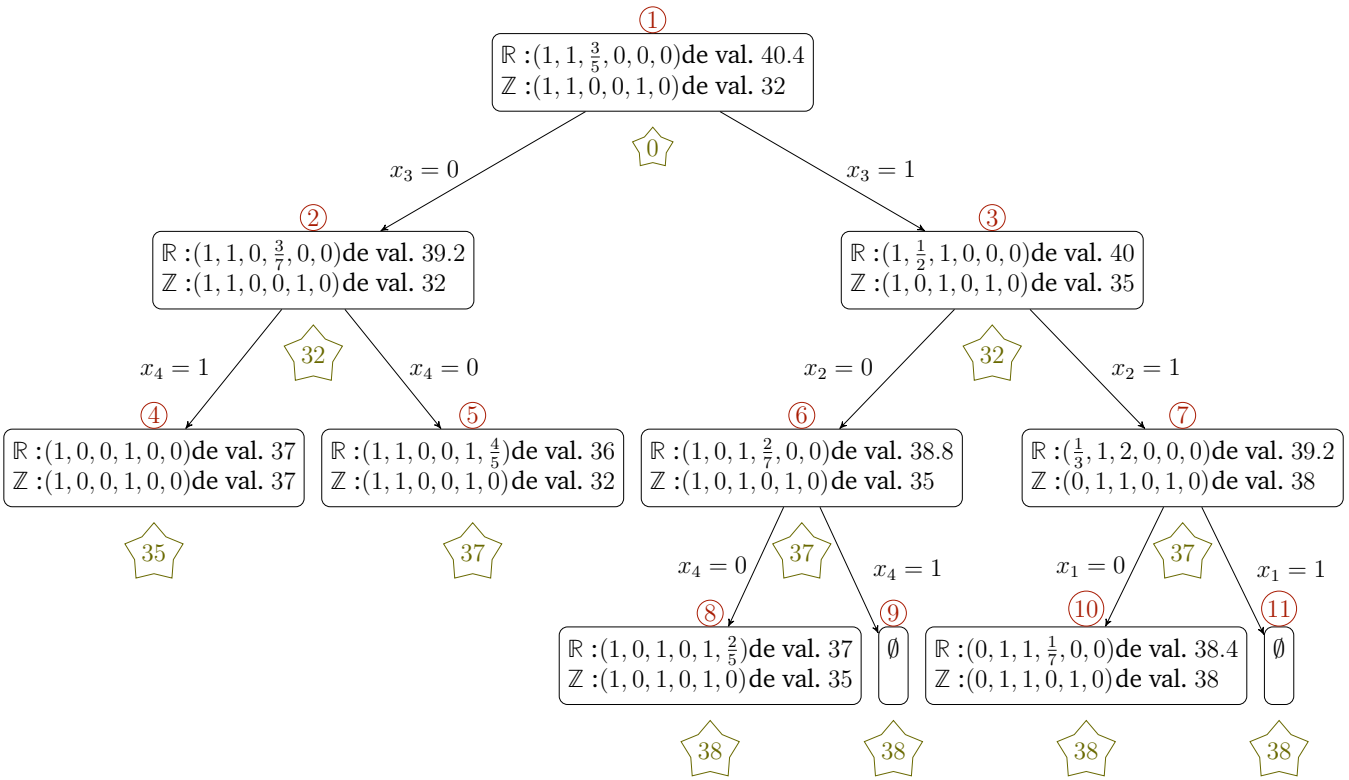


FIGURE 8 – Exécution de l'algorithme 6 sur l'instance exemple

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 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 faire tourner 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 faire tourner 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 2.21

On remarque que le majorant sur la valeur optimale dont on dipsose 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 $\leqslant 40$ ♣).

♣. En réalité, sachant que la valeur opt de la solution optimale de l'instance de départ est entière, on avait dès le

Exercice de cours 2.22

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.
- 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 2.23

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.

---

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).

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

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 2.24

> On a présenté ici le Branch-and-bound comme une méthode de résolution exacte, mais on peut auusi 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 couteuse.