TP 7 INFO — Algorithmes gloutons

Partie I — Problèmes d'optimisation

I.A - Généralités

Un problème d'optimisation consiste à trouver une solution optimale (selon un critère donné) parmi toutes les solutions d'un problème. Plus formellement, si $\mathscr{S}$ est un ensemble fini décrivant toutes les solutions d'un problème, alors on cherche à minimiser (ou maximiser) une fonction $\varphi : \mathscr{S} \to \mathbb{R}$ et à déterminer une solution $s \in \mathscr{S}$ permettant d'atteindre ce minimum (ou maximum).

Exemple 1 : Le problème du voyageur de commerce est un problème d'optimisation : il s'agit, étant donné n villes et leurs distances par paire, de trouver le chemin le plus court qui passe exactement une fois par chaque ville et revient à la ville de départ.

Une solution élémentaire serait de parcourir exhaustivement les éléments $s \in \mathscr{S}$ pour déterminer la valeur minimale (ou maximale) de $\varphi(s)$. Cette méthode est inefficace en général : le cardinal de $\mathscr{S}$ étant souvent exponentiel par rapport aux paramètres du problème, il en est de même pour la complexité temporelle de l'algorithme permettant de parcourir exhaustivement les éléments de $\mathscr{S}$.

Exemple 2 : Pour le problème du voyageur de commerce avec n villes, on peut vérifier, en tenant compte des symétries du problème, que l'approche exhaustive nécessite de tester $\dfrac{(n-1)!}{2}$ chemins candidats possibles.

Remarque 1 : Certains problèmes d'optimisation peuvent se résoudre de manière plus efficace (qu'avec une recherche exhaustive) en utilisant la programmation dynamique. Cette méthode sera étudiée en seconde année.

I.B - Résolution approchée via un algorithme glouton

Lorsque l'ensemble $\mathscr{S}$ est trop grand, il n'est pas possible de parcourir exhaustivement tous ses éléments. Dans ce cas, on peut utiliser un algorithme glouton, i.e. une stratégie (ou une heuristique) permettant de faire des choix rapides pour construire une solution, mais qui ne sera pas nécessairement optimale.

En général, un algorithme glouton est rapide, mais en contre-partie, la solution qu'il construit n'est pas nécessairement optimale. Cependant, si l'heuristique est suffisamment bien choisie, on peut espérer obtenir une solution « presque » optimale.

Exemple 3 : Pour le problème du voyageur de commerce, on peut utiliser l'algorithme glouton suivant :

- on commence par choisir aléatoirement la première ville à visiter.
- pour les étapes suivantes, le voyageur se déplace vers la ville non visitée la plus proche de sa ville actuelle ;
- une fois la dernière ville visitée, on retourne à la ville de départ.

Remarque 2 : Pour construire une solution approchée d'un problème d'optimisation, différentes stratégies gloutonnes sont possibles.

Dans la suite de ce TP, on étudie quelques exemples de problème d'optimisation que l'on peut résoudre de manière approchée en utilisant des stratégies gloutonnes.

Partie II Problème du rendu de monnaie

II.A - Description du problème

Dans cette partie, on étudie le problème du rendu de monnaie. Étant donné un système de monnaie (pièces et billets), on cherche le nombre minimal de pièces et de billets nécessaire pour réaliser une somme donnée.

Plus formellement, le système monétaire est représenté par un ensemble $\mathscr{M} = \{m_1, \dots, m_n\}$ où $m_1, \dots, m_n$ sont des entiers naturels non nuls distincts que l'on suppose rangés dans l'ordre croissant. Pour être certain que toute valeur entière s'écrit comme une somme d'éléments de $\mathscr{M}$, on impose également que $m_1 = 1$. L'ensemble des manières de rendre la valeur $V \in \mathbb{N}$ est décrit par l'ensemble

$$\mathscr{S} = \left\{ (x_1, \dots, x_n) \in \mathbb{N}^n \;\middle|\; \sum_{k=1}^{n} x_k m_k = V \right\}.$$

Le problème d'optimisation revient ainsi à minimiser la fonction $\varphi : \mathscr{S} \to \mathbb{N}$ définie par

$$\varphi : (x_1, \dots, x_n) \mapsto \sum_{k=1}^{n} x_k.$$

Exemple 4 : Considérons le système monétaire $\mathscr{M} = \{m_1, m_2, m_3\} = \{1, 3, 4\}$ et la somme $V = 6$. Dans ce cas simple, on peut écrire explicitement l'ensemble $\mathscr{S}$ de toutes les possibilités pour rendre la valeur 6 :

$$V = 6 \times m_1, \qquad V = 3 \times m_1 + 1 \times m_2, \qquad V = 2 \times m_1 + 1 \times m_3, \qquad V = 2 \times m_2.$$

On en déduit que

$$\mathscr{S} = \big\{(6,0,0), (1,3,0), (2,0,1), (0,2,0)\big\},$$

donc le nombre minimal de pièces dans ce cas est 2 et il y a une unique solution optimale qui est $(0,2,0)$.

II.B - Résolution approchée via une stratégie gloutonne

Lorsque les paramètres du problème sont grands, il n'est plus possible de tester toutes les possibilités pour rendre le montant V. Dans ce cas, on peut construire une solution approchée en utilisant la stratégie gloutonne suivante : tant qu'il reste quelque chose à rendre, choisir la plus grande valeur de pièce inférieure ou égale au montant à rendre.

Exemple 5 : Reprenons le système monétaire $\mathscr{M} = \{m_1, m_2, m_3\} = \{1, 3, 4\}$ avec la somme $V = 6$.

1) On commence par rendre $m_3 = 4$ qui est la plus grande valeur de $\mathscr{M}$ inférieure ou égale à $V = 6$.
2) Il reste $V = 2$ à rendre. On choisit $m_1 = 1$ qui est la plus grande valeur de $\mathscr{M}$ inférieure ou égale à $V = 2$.
3) Il reste $V = 1$ à rendre. On choisit $m_1 = 1$ qui est la plus grande valeur de $\mathscr{M}$ inférieure ou égale à $V = 1$.

Ainsi, la solution obtenue en utilisant la stratégie gloutonne est $(x_1, x_2, x_3) = (2, 0, 1)$ qui utilise 3 pièces (alors que la solution optimale n'en nécessitait que 2).

Question 1 : Écrire une fonction `rendu_monnaie_glouton(M:list, V:int) -> list` prenant en entrée une liste d'entiers `M` représentant le système monétaire et qui renvoie, sous forme de liste, la solution obtenue $(x_1, \dots, x_n)$ pour rendre la valeur `V` en mettant en œuvre la stratégie gloutonne décrite ci-dessus.

Question 2 : Évaluer la complexité temporelle et spatiale de votre fonction `rendu_monnaie_glouton`.

Remarque 3 : Pour le système monétaire des euros $\mathscr{M} = \{1, 2, 5, 10, 20, 50, 100, 200\}$, on peut démontrer que l'algorithme glouton décrit ci-dessus fournit toujours la solution optimale.

Partie III Problème du sac à dos

III.A - Description du problème

Étant donné une liste finie d'objets ayant chacun un poids et une valeur, le problème du sac à dos consiste à remplir un sac à dos en maximisant la valeur de son contenu tout en respectant le poids maximal qu'il peut supporter.

Plus formellement, les objets sont représentés par des couples $(p_1, v_1), \dots, (p_n, v_n) \in \mathbb{N}^* \times \mathbb{N}^*$ indiquant respectivement leur poids et leur valeur. Si on désigne par $C \in \mathbb{N}$ la capacité du sac, l'ensemble des manières de le remplir est décrit par

$$\mathscr{S} = \left\{ I \subset [\![1, n]\!] \;\middle|\; \sum_{i \in I} p_i \leqslant C \right\}.$$

Le problème du sac à dos revient ainsi à maximiser la fonction $\varphi : \mathscr{S} \to \mathbb{N}$ définie par

$$\varphi : I \mapsto \sum_{i \in I} v_i.$$

Exemple 6 : On considère un sac à dos de capacité $C = 10$ et les $n = 5$ objets

$$(p_1, v_1) = (2,2), \quad (p_2, v_2) = (3,4), \quad (p_3, v_3) = (3,7), \quad (p_4, v_4) = (5,8), \quad (p_5, v_5) = (7,11).$$

On peut remplir le sac à dos avec les objets indiqués par $I = \{1,5\}$: on obtient une valeur totale de $v_1 + v_5 = 13$ pour un poids total de $p_1 + p_5 = 9 \leqslant C$. Cependant, on peut vérifier que la valeur 9 n'est pas maximale.

En Python, on indiquera les objets considérés avec une liste comme ci-dessous.

```
lst_objets = [(2,2), (3,4), (3,7), (5,8), (7,11)]
```

III.B - Résolution approchée via différentes stratégies gloutonnes

Nous allons résoudre ce problème de manière approchée en utilisant différentes stratégies gloutonnes.

1) **Stratégie gloutonne 1 :** tant qu'il est possible d'ajouter des objets dans le sac en respectant sa capacité, on ajoute l'objet dont la valeur est la plus grande. On espère ainsi maximiser la valeur totale dans le sac.
2) **Stratégie gloutonne 2 :** tant qu'il est possible d'ajouter des objets dans le sac en respectant sa capacité, on ajoute l'objet dont le poids est le plus petit. On espère ainsi maximiser le nombre d'objets dans le sac et peut-être par là-même la valeur totale dans le sac.
3) **Stratégie gloutonne 3 :** tant qu'il est possible d'ajouter des objets dans le sac en respectant sa capacité, on ajoute l'objet dont le rapport valeurs / poids est le plus grand. On espère que cela soit un compromis intéressant entre les deux stratégies précédentes.

Pour trier les objets selon un certain critère (une clé), on peut utiliser la syntaxe ci-dessous.

```
# On définit la clé de tri qui est une fonction
# Dans cet exemple, Python va comparer la valeur des objets
def cle_de_tri(objet):
    poids, valeur = objet
    return(valeur)

# La syntaxe ci-dessous permet d'obtenir une version triée lst de la liste lst_obj
# où les éléments sont rangés dans l'ordre croissant pour la clé cle_de_tri
lst = sorted(lst_objets, key = cle_de_tri)
```

Question 3 : Écrire une fonction `sac_a_dos_glouton1(lst_objets:list, C:int) -> list` qui renvoie la liste des objets à mettre dans le sac en mettant en œuvre la stratégie gloutonne 1 décrite ci-dessus.

Question 4 : Écrire une fonction `sac_a_dos_glouton2(lst_objets:list, C:int) -> list` qui renvoie la liste des objets à mettre dans le sac en mettant en œuvre la stratégie gloutonne 2 décrite ci-dessus.

Question 5 : Écrire une fonction `sac_a_dos_glouton3(lst_objets:list, C:int) -> list` qui renvoie la liste des objets à mettre dans le sac en mettant en œuvre la stratégie gloutonne 3 décrite ci-dessus.

Question 6 : Évaluer la complexité temporelle et spatiale de vos trois fonctions ci-dessus.

Question 7 : Évaluer la qualité des trois stratégies gloutonnes précédentes en les testant sur différents exemples.

Partie IV Problème du voyageur de commerce

Dans cette partie, nous allons résoudre de manière approchée le problème du voyageur de commerce avec une stratégie gloutonne en considérant quelques villes sur Terre. Dans la suite, on considère qu'une ville est représentée par un tuple de trois éléments : le premier élément est une chaîne de caractères donnant son nom, le second élément est un flottant donnant sa latitude φ en radian et le troisième élément est un flottant donnant sa longitude λ en radian.

Si deux points ont pour coordonnées géographiques respectives (φ_A, λ_A) et (φ_B, λ_B), alors la distance à vol d'oiseau entre ces deux points (en kilomètre) est donnée par la formule

$$d\big((\varphi_A,\lambda_A),(\varphi_B,\lambda_B)\big) = R \times \text{Arccos}\big(\sin(\varphi_A)\sin(\varphi_B) + \cos(\varphi_A)\cos(\varphi_B)\cos(\lambda_B - \lambda_A)\big) \quad \text{avec} \quad R = 6\,378,137.$$

Question 8 : Écrire une fonction `d(ville_A:tuple, ville_B:tuple) -> float` prenant en entrée deux tuples représentant deux villes et qui renvoie la distance entre ces deux villes.

Question 9 : Écrire une fonction `voyageur_commerce_glouton(lst_villes:list) -> list` prenant en entrée une liste de villes sur terre et renvoyant la liste ordonnée des villes à visiter en appliquant la stratégie gloutonne décrite dans l'introduction du TP.

Question 10 : Évaluer la complexité temporelle et spatiale de la fonction précédente.

Récupérez le fichier `villes_francaises.csv` dans la rubrique informatique sur le site `http://vonbuhren.free.fr` qui contient les coordonnées géographiques de différentes villes françaises.

Question 11 : Appliquez votre fonction précédente à la liste dont les éléments sont les villes présentes dans le fichier `villes_francaises.csv`.