Algorithmes gloutons

À la fin de ce chapitre, je sais :

- ☞ expliquer le principe d'un algorithme glouton
- ☞ citer des cas d'utilisation classiques de ce principe
- ☞ coder un algorithme glouton en Python

A Principe

■ **Définition 1 — Optimisation.** Un problème d'optimisation $\mathcal{P}$ nécessite de déterminer les conditions dans lesquelles ce problème présente une caractéristique optimale au regard d'un critère.

■ **Exemple 1 — Problèmes d'optimisation.** La plupart des problèmes de tous les jours sont des problèmes d'optimisation :

- comment répartir équitablement des tâches selon certains critères ?
- comment choisir le plus court chemin pour aller d'un point à un autre ?
- comment choisir ses actions pour optimiser un portefeuille et son rendement ?
- comment choisir le régime moteur pour économiser un maximum de carburant ?
- comment choisir des articles dans un supermarché en respectant un budget et d'autres contraintes simultanément ?
- comment faire ses valises en emportant à la fois le plus d'affaires possibles et le plus de valeur possible au global ?

■ **Définition 2 — Algorithme glouton.** Un algorithme glouton décompose un problème en sous-problèmes et le résout en :

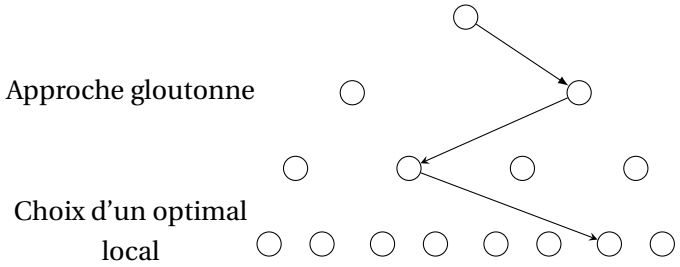

FIGURE 1 – Étape de résolution d'un problème par décomposition en sous-problèmes et approche gloutonne.ă

1. construisant une solution partielle en effectuant à chaque étape le meilleur choix local,
2. espérant que ces choix locaux conduiront à un résultat global optimal.

Vocabulary 1 — Greedy algorithms ⇜⇝ les algorithmes gloutons

La figure 1 illustre le fonctionnement d'un algorithme glouton. Un problème de décomposition devient une suite de choix optimaux localement.

R La plupart du temps, un algorithme glouton est appliqué à un problème d'optimisation. Le résultat n'est pas toujours optimal. Mais l'espoir fait vivre : certains algorithmes gloutons obtiennent une solution optimale! Comme ils sont souvent assez simples à implémenter par rapport aux autres algorithmes d'optimisation, ils représentent une solution précieuse.

■ **Définition 3 — Glouton optimal.** On dit qu'un algorithme glouton est optimal s'il produit une solution optimale au problème d'optimisation associé.

■ **Exemple 2 — Algorithmes gloutons optimaux.** Parmi les algorithmes au programme, il existe des algorithmes gloutons optimaux :

- Dijkstra (plus court chemin),
- Prim et Kruskal (arbres recouvrants),
- codage d'Huffmann.

B Modélisation

On considère un ensemble $\mathcal{E}$ d'éléments parmi lesquels on doit faire des choix pour optimiser le problème $\mathcal{P}$. On construit une solution $\mathcal{S}$ séquentiellement via un algorithme glouton en suivant la procédure décrite sur l'algorithme 1. Il ne reste plus qu'à préciser, selon le problème considéré :

- le choix de l'élément optimal localement dans $\mathcal{E}$,
- le test d'une solution pour savoir si celle-ci est complète ou partielle,
- l'ajout d'un élément à une solution.

Algorithme 1 Principe d'un algorithme glouton

```
1 : Fonction GLOUTON(E)                          ▷ E un ensemble d'éléments
2 :     S ← ∅
3 :     tant que S pas complète et E pas vide répéter
4 :         e ← CHOISIR_UN_ÉLÉMENT(E)            ▷ le meilleur localement!
5 :         si l'ajout de e à S est une solution possible alors
6 :             S ← S + e
7 :         Retirer e de E                       ▷ Si pas déjà fait en 4
8 :     renvoyer S
```

a Modélisation de l'occupation de la place au port

On désigne par $\mathcal{E}$ l'ensemble des demandes des clients. Pour toute demande $e \in \mathcal{E}$, on a la possibilité d'accéder à la date de début $d(e)$ de la demande ainsi qu'à la date de fin $f(e)$. On cherche donc à trouver un sous-ensemble de $\mathcal{E}$ constitué de demandes compatibles et de cardinal maximum.

On dénote l'intervalle de temps d'occupation associé à une demande e par $[d(e), f(e)]$. La compatibilité de deux demandes e_i et e_j peut alors être formalisée ainsi :

$$]d(e_i), f(e_i)[\cap]d(e_j), f(e_j)[= \emptyset \tag{1}$$

L'algorithme glouton 2 permet de résoudre ce problème de planning.

Algorithme 2 Réservation d'une place au port de Brest

```
1 : Fonction GÉRER_LA_PLACE(E)                   ▷ E un ensemble de demandes e_i
2 :     S ← ∅
3 :     Trier E pas ordre de valeurs f(e_i) croissantes   ▷ Dates de fin croissantes
4 :     Ajouter à S la demande qui se termine le plus tôt.
5 :     tant que E pas vide répéter
6 :         e ← retirer l'élément de E qui se termine le plus tôt
7 :         s ← dernier élément ajouté dans S
8 :         si d(e) ⩾ f(s) alors                 ▷ Est-ce une solution?
9 :             Ajouter e à S
10 :    renvoyer S
```

Cet algorithme est bien glouton car :

1. la construction de l'ensemble $\mathcal{S} = \{s_1, ..., s_k\}$ s'effectue de manière séquentielle,

2. le choix effectué à chaque tour de boucle est le meilleur en terme de compatibilité : les demandes peuvent éventuellement s'enchaîner grâce à la ligne 8.

L'ensemble $\mathcal{S}$ étant constitué de demandes compatibles, l'algorithme aboutit à une solution. Cependant, on peut se demander si celle-ci est optimale, c'est à dire de cardinal maximum : a-t-on satisfait un maximum de clients ? Ne peut-on faire mieux ?

b Preuve de l'optimalité --→ HORS PROGRAMME

Théorème 1 — L'algorithme 2 aboutit à une solution optimale.

Démonstration par induction sur les ensembles solutions $\mathcal{S} = \{s_1, s_2, \ldots, s_j\}$, $j \in [\![1, k]\!]$. On suppose qu'on peut ordonner les ensembles $\mathcal{E}$ et $\mathcal{S}$ par date de fin croissante et qu'il y a une date à laquelle on a démarré le service de location [1]. Pour un ensemble de demandes $\Delta = \{\delta_1, \ldots, \delta_r\}$ ainsi ordonné, les demandes $\delta_1, \ldots, \delta_r$ sont compatibles si et seulement si :

$$d(\delta_1) < f(\delta_1) \leqslant d(\delta_2) < f(\delta_2) \leqslant \ldots \leqslant d(\delta_r) < f(\delta_r) \tag{2}$$

On procède en deux temps en montrant :

1. d'abord qu'il existe une solution optimale $\mathcal{O} = \{o_1, \ldots, o_m\}$ telle que les premiers éléments de $\mathcal{S} = \{s_1, \ldots, s_k\}$ coïncident avec ceux de $\mathcal{O}$,
2. puis que $\mathcal{S} = \mathcal{O}$.

Initialisation : pour $j = 1$, on a $\mathcal{S} = s_1$, s_1 étant la demande qui se termine le plus tôt. Par rapport à la solution optimale $\mathcal{O} = \{o_1, ..., o_m\}$, on a nécessairement $f(s_1) \leqslant f(o_1)$ et donc :

$$d(s_1) < f(s_1) \leqslant d(o_2) < f(o_2) \leqslant \ldots \leqslant d(o_m) < f(o_m) \tag{3}$$

On en conclut que $\{s_1, o_2, \ldots, o_m\}$ est compatible avec $\mathcal{O}$. Pour $j = 1$ et $\mathcal{S}$ est donc optimale.

Pour l'hérédité, on suppose que la solution $\mathcal{S} = \{s_1, \ldots, s_j\}$ est compatible avec $\mathcal{O} = \{o_1, \ldots, o_m\}$, c'est à dire que : $\{s_1, s_2, \ldots, s_j, o_{j+1}, \ldots o_m\}$ est optimale. L'algorithme 2 assure que s_{j+1} est choisi dans $\mathcal{E} \setminus \{s_1, \ldots, s_j\}$, de telle manière que la date de fin est minimale. D'après notre hypothèse, o_{j+1} est choisi dans $\mathcal{E} \setminus \{s_1, \ldots, s_j\}$, de telle manière que la date de fin est minimale. Donc $f(s_{j+1}) \leqslant f(o_{j+1})$. On en conclut que $\{s_1, s_2, \ldots, s_j, s_{j+1}, o_{j+2}, \ldots o_m\}$ est compatible avec $\mathcal{O}$ et donc optimale.

Par induction, on peut donc affirmer que $\mathcal{S}$ est optimale, quelque soit $j \in [\![1, k]\!]$. Est-ce que les deux solutions optimales $\mathcal{S}$ et $\mathcal{O}$ coïncident ? C'est à dire, est que $k = m$? Si ce n'était pas le cas, c'est à dire $k > m$, alors on pourrait choisir dans $\mathcal{E} \setminus \{s_1, \ldots, s_k\}$ une demande dont la date de fin serait compatible avec la solution optimale. Cette solution ne serait donc plus optimale, ce qui est une contradiction. ■

Cet algorithme glouton est donc optimal, mais cela est essentiellement dû aux contraintes qu'on a mise sur l'optimisation : tous les clients ne seront pas satisfaits. On cherche juste à en satisfaire un maximum.

1. sous-entendu, on ne pourra pas choisir une date plus petite que celle-ci pour le début et la fin de la location.

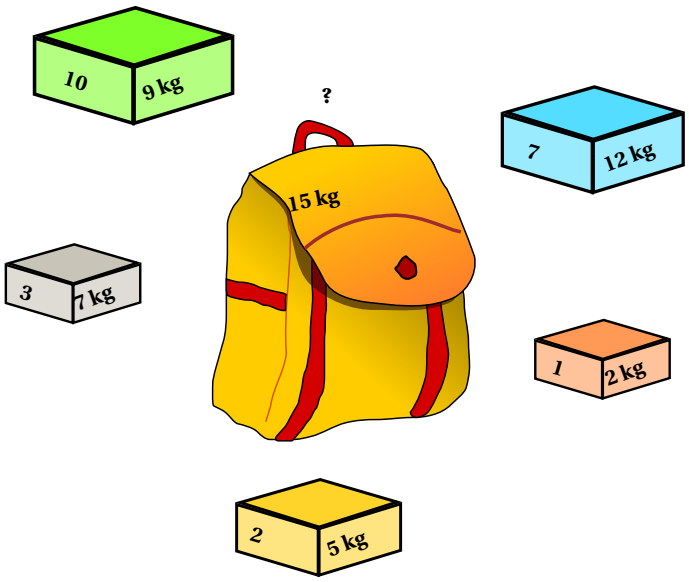

FIGURE 2 – Illustration du problème du sac à dos (d'après Wikipedia). On a cinq objets de poids 9, 12, 2, 7 et 5 kg et de valeur 10, 7, 1, 3 et 2 . Le poids total admissible dans le sac est 15kg.

C Exemple du sac à dos

On cherche à remplir un sac à dos comme indiqué sur la figure 2. Chaque objet que l'on peut insérer dans le sac est **insécable**[2] et possède une valeur et un poids connus. On cherche à maximiser la valeur totale emportée dans la sac à dos tout en limitant[3] le poids à π.

Vocabulary 2 — Knapsack problem ⇜⇝ Le problème du sac dos.

On peut chercher à résoudre le problème du sac à dos de manière gloutonne en utilisant un algorithme glouton (cf. algorithme 3).

En terme de complexité, l'algorithme 3 est plutôt intéressant puisqu'en $O(n)$. Cependant, la solution trouvée n'est pas nécessairement optimale : cet algorithme est donc un point de départ mais pas la solution définitive à ce problème.

R Intuitivement, on comprend bien qu'on ne pourra trouver de solution optimale que si on peut compléter le sac à dos au maximum. Il faut donc que les objets qui restent à mettre dans le sac après le plus valué soient :

- de la bonne taille, ce qui revient à invoquer la chance,
- ou sécable afin de se conformer à l'espace restant et l'optimiser. Or on a choisi des objets insécables.

2. Soit on le met dans le sac, soit on ne le met pas. Mais on ne peut pas en mettre qu'une partie.
3. On accepte un poids total inférieur ou égal à π.

Algorithme 3 Problème du sac à dos

```
1 : Fonction SAC_À_DOS(ℰ, π)                          ▷ ℰ = {(v_1, p_1), ..., (v_n, p_n)}
2 :     𝒮 ← ∅
3 :     p_total ← 0
4 :     v_total ← 0
5 :     Trier ℰ pas ordre de valeurs v_i décroissantes          ▷ le choix sera facile
6 :     tant que p_total ⩽ π et ℰ pas vide répéter
7 :         v, p ← retirer l'élément de ℰ le plus valué        ▷ choix de v maximale
8 :         si p_total + p ⩽ π alors                           ▷ Est-ce une solution ?
9 :             Ajouter (v, p) à 𝒮
10 :    renvoyer 𝒮
```

D'une manière plus générale, le problème du sac à dos reflète un problème d'allocation de ressources pour lequel le temps (ou le budget) est fixé et où l'on doit choisir des éléments indivisibles parmi un ensemble tâches (ou de projets).

D Gloutonnerie et dynamisme

Tenter sa chance est le plus souvent payant ! Si un algorithme glouton donne une solution rapidement alors que l'algorithme donnant la solution optimale est de complexité exponentielle $O(2^n)$ alors tenter sa chance en *gloutonnant* permet souvent de progresser vers une solution acceptable.

R La programmation dynamique qui sera étudiée au semestre trois permet de résoudre les problèmes d'optimisation sur lesquels butent certains algorithmes gloutons. Elle est souvent pertinente lorsque les sous-problèmes générés se recoupent, cas pour lesquels les algorithmes gloutons échouent.