

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 Problème d'optimisation

■ **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.

R Optimiser, c'est donc construire une solution S en choisissant parmi les éléments d'un ensemble \mathcal{E} ceux qui génèrent le meilleur résultat selon le critère de l'optimisation.

■ **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 possible et le plus de valeur possible au total?

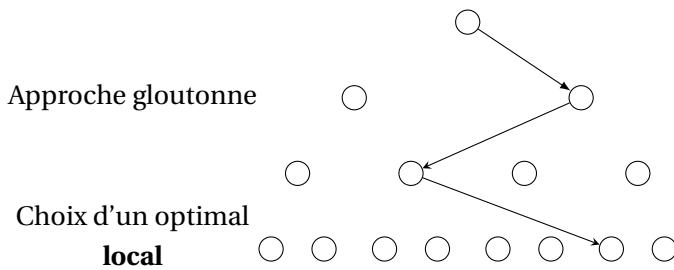


FIGURE 1 – Étape de résolution d'un problème d'optimisation par décomposition en sous-problèmes et approche gloutonne.

B Algorithmes gloutons

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

1. en construisant une solution partielle admissible en effectuant à chaque étape le **meilleur choix local**,
2. en **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 dans un graphe),
- Prim et Kruskal (arbres recouvrants d'un graphe),
- codage d'Huffman (compression de données).

C Modélisation

On considère un ensemble \mathcal{E} d'éléments parmi lesquels on doit faire une sélection pour optimiser la solution \mathcal{S} à un problème d'optimisation \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 du meilleur élément de \mathcal{E} , **l'optimum local**,
- le test d'une solution pour savoir si celle-ci est une solution complète ou partielle du problème \mathcal{P} ,
- l'ajout d'un élément à une solution.

Algorithme 1 Principe d'un algorithme glouton

```

1: Fonction GLOUTON( $\mathcal{E}, \mathcal{P}$ )                                 $\triangleright \mathcal{E}$  un ensemble d'éléments,  $\mathcal{P}$  le problème
2:    $\mathcal{S} \leftarrow \emptyset$                                           $\triangleright$  La solution de  $\mathcal{P}$  à construire
3:   tant que  $\mathcal{S}$  n'est pas une solution complète de  $\mathcal{P}$  et que  $\mathcal{E}$  n'est pas vide répéter
4:      $e \leftarrow \text{CHOISIR\_LE\_MEILLEUR\_ÉLÉMENT\_LOCAL}(\mathcal{E})$            $\triangleright$  l'optimum local!
5:     si l'ajout de  $e$  à  $\mathcal{S}$  est une solution (partielle ou complète) de  $\mathcal{P}$  alors
6:        $\mathcal{S} \leftarrow \mathcal{S} \cup \{e\}$                                           $\triangleright$  On opère le choix glouton!
7:      $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$                                           $\triangleright$  Si cela n'a pas déjà été fait en 4
8:   renvoyer  $\mathcal{S}$ 

```

a Exemple de l'occupation de la place au port

Un port de plaisance gère l'occupation d'une place vacante et ouverte à la réservation pour une durée limitée. Certains plaisanciers sont de passage et font des réservations. L'objectif fixé est de sélectionner **un maximum de réservations compatibles** afin de satisfaire un maximum de clients.

On désigne par \mathcal{E} l'ensemble des demandes des clients. Pour toute demande $C \in \mathcal{E}$, on a la possibilité d'accéder à la date de début $d(C)$ de la demande ainsi qu'à la date de fin $f(C)$. 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 C par $[d(C), f(C)]$. La compatibilité de deux demandes C_i et C_j peut alors être formalisée ainsi :

$$[d(C_i), f(C_i)] \cap [d(C_j), f(C_j)] = \emptyset \quad (1)$$

Un exemple de réservations est donné sur la figure 2.

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

Cet algorithme est bien glouton car :

1. la construction de l'ensemble $\mathcal{S} = \{s_1, \dots, s_k\}$ s'effectue de manière séquentielle,
2. le choix effectué à chaque tour de boucle est le meilleur en termes de compatibilité : les demandes peuvent éventuellement s'enchaîner grâce à la ligne ??.

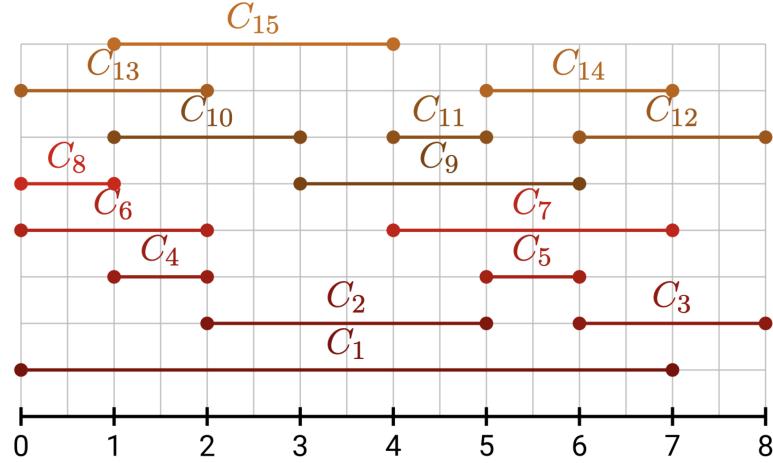


FIGURE 2 – Exemple de réservations de place au port correspondant à l'ensemble $\{[0,7], [2,5], [6,8], [1,2], [5,6], [0,2], [4,7], [0,1], [3,6], [1,3], [4,5], [6,8], [0,2], [5,7], [1,4]\}$

Algorithme 2 Réservation d'une place au port (Version simplifiée)

```

1: Fonction GÉRER_LA_PLACE( $\mathcal{E}$ )
2:    $\mathcal{S} \leftarrow \emptyset$ 
3:   Trier  $\mathcal{E}$  par ordre croissant des dates de fin  $f(C_i)$ 
4:   date_limite  $\leftarrow -\infty$ 
5:   pour  $C \in \mathcal{E}$  répéter                                 $\triangleright$  On parcourt la liste triée
6:     si  $d(C) \geqslant \text{date\_limite}$  alors
7:        $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$ 
8:       date_limite  $\leftarrow f(C)$ 
9:   renvoyer  $\mathcal{S}$ 

```

Pour l'exemple de la figure 2, cet algorithme trouve $\{[0, 1], (1, 2), (2, 5), (5, 6), (6, 8)\}$, c'est-à-dire C_8, C_4, C_2, C_5 et C_3 . L'ensemble \mathcal{S} étant constitué de demandes partiellement compatibles, l'algorithme aboutit donc 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

R Dans un premier temps, on peut remarquer que l'ensemble \mathcal{S} est un sous-ensemble de \mathcal{E} qui est lui-même un ensemble fini. C'est pourquoi, il existe nécessairement une solution de cardinal maximum. En effet, toute partie non vide finie de \mathbb{N} est majorée et elle admet un plus grand élément.

L'algorithme 2 aboutit à une solution optimale.

Démonstration par induction sur les ensembles solutions $\mathcal{S} = \{s_1, s_2, \dots, s_j\}, j \in \llbracket 1, k \rrbracket$. 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 location¹. Pour un ensemble de demandes $C = \{c_1, \dots, c_r\}$ ainsi ordonné, les demandes c_1, \dots, c_r sont compatibles si et seulement si :

$$d(c_1) < f(c_1) \leq d(c_2) < f(c_2) \leq \dots \leq d(c_r) < f(c_r) \quad (2)$$

On procède en deux temps en montrant :

1. d'abord qu'il existe une solution optimale $\mathcal{O} = \{o_1, \dots, o_m\}$ telle que les premiers éléments de $\mathcal{S} = \{s_1, \dots, s_k\}$ 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 $\mathcal{O} = \{o_1, \dots, o_m\}$, on a nécessairement $f(s_1) \leq f(o_1)$ et donc :

$$d(s_1) < f(s_1) \leq d(o_2) < f(o_2) \leq \dots \leq d(o_m) < f(o_m) \quad (3)$$

On en conclut que $\{s_1, o_2, \dots, o_m\}$ est compatible avec \mathcal{O} . Pour $j = 1$, la propriété est vérifiée et \mathcal{S} est compatible avec une solutions optimale.

Hérédité : on suppose que la solution $\mathcal{S} = \{s_1, \dots, s_j\}$ est compatible avec $\mathcal{O} = \{o_1, \dots, o_m\}$, c'est à dire que : $\{s_1, s_2, \dots, s_j, o_{j+1}, \dots, o_m\}$ est optimale. s_{j+1} est choisie minimale parmi toutes les tâches compatibles avec s_j . Or, comme $f(s_j) \leq f(o_j)$, l'ensemble des tâches compatibles avec s_j inclut l'ensemble des tâches compatibles avec o_j .

D'après notre hypothèse, o_{j+1} est choisi dans $\mathcal{E} \setminus \{s_1, \dots, s_j\}$, de telle manière que la date de fin est minimale. Donc $f(s_{j+1}) \leq f(o_{j+1})$. On en conclut que $\{s_1, s_2, \dots, s_j, s_{j+1}, o_{j+2}, \dots, o_m\}$ est compatible avec \mathcal{O} et donc optimale.

Par induction, on peut donc affirmer que \mathcal{S} est optimale, quel que soit $j \in \llbracket 1, k \rrbracket$. Est-ce que les deux solutions optimales \mathcal{S} et \mathcal{O} coïncident ? Est-ce que $k = m$? Si ce n'était pas le cas, c'est à dire $k > m$, alors on pourrait choisir dans $\mathcal{E} \setminus \{s_1, \dots, s_k\}$ une demande dont la date de fin serait

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

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 mises sur l'optimisation : **tous les clients ne seront pas satisfaits. On cherche juste à en satisfaire un maximum.**

D Exemple du sac à dos

On cherche à remplir un sac à dos comme indiqué sur la figure 3. Chaque objet que l'on peut insérer dans le sac est **insécable**² et possède une valeur entière (\mathbb{E}) et un poids entier connus (kg). On cherche à maximiser la valeur totale emportée dans le sac à dos tout en limitant³ le poids à π kg.

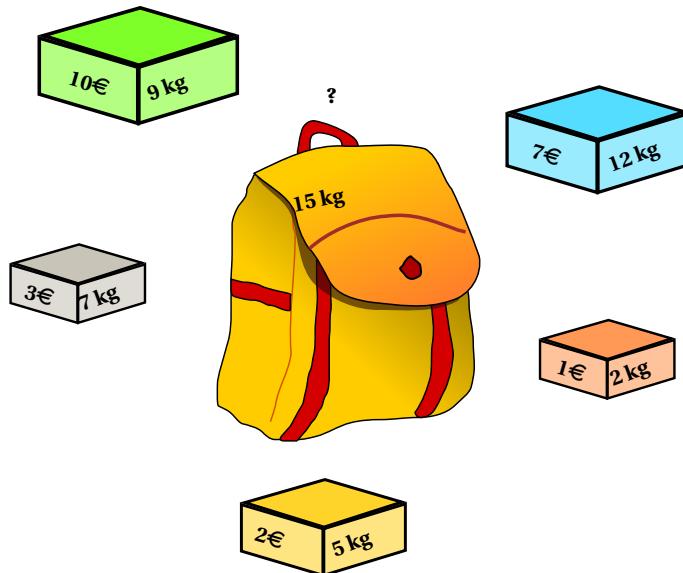


FIGURE 3 – Illustration du problème du sac à dos (d'après Wikipedia). On a cinq objets de poids 9, 12, 2, 7 et 5 kg et de valeur 10, 7, 1, 3 et 2 . Le poids total admissible dans le sac est 15kg.



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 termes de complexité, l'algorithme 3 est plutôt intéressant : sa complexité est la somme de celle du tri et de la boucle TANT QUE soit $O(n \log n + n)$. Cependant, la solution trouvée n'est

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( $\mathcal{E}, \pi$ )                                 $\triangleright \mathcal{E} = \{(v_1, p_1), \dots, (v_n, p_n)\}$ 
2:    $\mathcal{S} \leftarrow \emptyset$ 
3:    $p_{total} \leftarrow 0$ 
4:    $v_{total} \leftarrow 0$ 
5:   Trier  $\mathcal{E}$  par ordre de valeurs  $v_i$  décroissantes            $\triangleright$  le choix sera facile
6:   tant que  $p_{total} < \pi$  et que  $\mathcal{E}$  non vide répéter
7:      $v, p \leftarrow$  retirer l'élément de  $\mathcal{E}$  le plus valué       $\triangleright$  choix de  $v$  maximale
8:     si  $p_{total} + p \leq \pi$  alors                          $\triangleright$  Est-ce une solution?
9:       Ajouter  $(v, p)$  à  $\mathcal{S}$ 
10:  renvoyer  $\mathcal{S}$ 

```

pas nécessairement optimale : cet algorithme est donc un point de départ mais pas la solution définitive à ce problème.

R 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).

R Une intuition naturelle consisterait à trier les objets selon leur rapport valeur/poids (ou densité massique). Cependant, cette heuristique échoue également à fournir une solution optimale dans le cas général. La contrainte d'intégrité des objets (**insécables**) empêche souvent l'algorithme d'utiliser la totalité de la capacité disponible : l'espace résiduel ne peut être optimisé. C'est donc la nature discrète du problème qui empêche la méthode gloutonne de fonctionner systématiquement. Notez que si l'on relâche cette contrainte (problème du sac à dos fractionnaire où les objets sont sécables), alors la stratégie gloutonne basée sur la densité devient prouvablement optimale.

E 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 pertinente lorsque les sous-problèmes générés se recoupent.