Chapitre 12

Algorithmes gloutons

Notions	Commentaires
Algorithme glouton fournissant une solution exacte.	On peut traiter comme exemples d'algorithmes exacts : codage de Huffman, sélection d'activité, ordonnancement de tâches unitaires avec pénalités de retard sur une machine unique.

Extrait de la section 4.3 du programme officiel de MP2I : « Décomposition en sous-problèmes ».

Sommaire

0 Généralités

Contexte : On s'intéresse dans ce cours à la résolution de problèmes dont la construction d'une solution peut-être décomposée comme une succession de choix (comme en retour sur trace !). Le problème que l'on étudie peut-être un problème d'existence, ou bien (plus fréquemment !) un problème d'optimisation.

0.0 Principe général

0.0.0 Définition

Définition 1.
La technique algorithmique dite **gloutonne** consiste à résoudre un problème de la manière suivante :

- On construit une solution graduellement, choix après choix. On ne reviendra jamais en arrière.
- À chaque étape, on fait le choix qui est localement optimal.

Exemple. Considérons SubsetSum, version problème d'optimisation : étant donné une liste ℓ d'entiers et $t \in \mathbb{N}$, quelle est la somme maximale d'éléments de ℓ qui soit inférieure à t ?

Une solution gloutonne est de parcourir la liste et de prendre chaque élément si on peut l'ajouter aux éléments déjà pris sans que cela ne dépasse t.

Figure 12.1 – Algorithme glouton pour SubsetSum avec $\ell = [3; 2; 2; 10; 4; 5]$ et $t = 20$

Remarque.

- L'exemple précédent montre qu'un algorithme glouton peut ne pas renvoyer une solution optimale (il y a une solution qui se somme à 19). En fait, c'est même souvent le cas : une succession de choix localement optimaux mène rarement à une solution globalement optimale !
- En fait, un glouton ne permet presque jamais de calculer une solution optimale, mais souvent une approximation correcte d'une solution optimale (cf MPI).
- La différence avec le retour sur trace est que ce dernier peut revenir en arrière pour remettre en question un choix précédent [1].

1. Cela signifie aussi que quand vous écrivez un "retour sur trace" récursif qui ne vérifie pas le résultat de l'appel récursif... vous ne remettez pas en cause le choix actuel s'il a été invalidé par les appels récursifs, et vous faites donc en fait un algorithme glouton.

0.0.1 Intérêt

Avantages	Inconvénients
Simple à concevoir Faible complexité temporelle Permet parfois d'approximer décemment une solution optimale. Les preuves de correction sont souvent (très) similaires.	Renvoie généralement une solution non optimale.

Table 12.1 – Avantages et inconvénients d'un algorithme glouton

0.1 Exemple : ordonnancement d'intervalle

On considère un gymnase qui organise une journée sportive dans sa grande salle. De nombreuses personnes veulent organiser une activité : elles requierent la salle sur un intervalle de temps donné. Deux activités ne peuvent pas avoir lieu en même temps. Comment organiser le plus d'activités possibles ?

Figure 12.2 – Problème du gymnase (ordonnancement d'intervalles)

Définition 2 (Ordonnancement d'intervalles).
Le problème d'ordonnancement d'intervalles ets le suivant :

Entrée : un ensemble $S = \{r_0, ..., r_{n-1}\}$ de n requêtes. Chaque requête r_i est un intervalle ouvert non-vide $]d_i; f_i[$ (avec $d_i < f_i$).

Tâche : Trouver la plus grande partie de S constituée d'ensembles deux à deux disjoints

Remarque.

- En anglais, ce problème s'appelle *Interval Scheduling*.
- On peut représenter le problème comme un graphe : les requêtes sont des noeuds, et deux noeuds sont reliés si leurs intervalles s'intersectent. Comme l'intersection est symétrique, le graphe est non-orienté. On veut alors trouver un ensemble de noeuds deux à deux non voisins : on appelle cela un *indépendant* du graphe.

  Il s'avère que trouver un plus grand indépendant dans un graphe quelconque correspond à un problème NP-Complet (donc a priori impossible à résoudre en temps polynomial[2])... sauf que dans notre gymnase, on ne se base pas sur n'importe quel graphe : c'est un graphe (d'intersection d')intervalles ! Peut-être existe-t-il un glouton ?

2. Sauf si P = NP. Mais je m'avance beaucoup sur la MPI là !

Idée gloutonne	Contre-exemple à l'optimalité

Table 12.2 – Des idées gloutonnes pour Interval Scheduling et des contre-exemples

0.2 Preuve d'optimalité

Proposition 3 (Preuve par échange).
Pour prouver qu'un algorithme glouton est correct (c'est à dire qu'il renvoie bien une solution optimale), on procède souvent ainsi :

1. Lemme d'échange : Montrer qu'il existe une solution optimale où le premier choix effectué est celui de l'algorithme glouton.
2. Récurrence : Montrer qu'on peut combiner ce premier choix glouton avec une solution optimale du sous-problème qu'il reste à résoudre en une solution optimale du problème. En déduire la correction de l'algorithme par récurrence.

Remarque.

- Le schéma de preuve proposé ci-dessus marche pour *beaucoup* d'algorithmes gloutons.
- La difficulté est généralement le lemme d'échange ; mais il ne faut pas oublier de conclure par récurrence pour prouver que la solution gloutonne est optimale !

Exemple. Prouvons la correction de l'algorithme glouton « prendre en priorité les requêtes se terminant en premier » pour l'ordonnancement d'intervalles. Considérons donc $\mathscr{R} = \{r_0, ..., r_{n-1}\}$ les requêtes, avec $r_i =]d_i; f_i[$; on les suppose triées par f_i croissant.

- Lemme d'échange : Montrons qu'il existe une solution optimale contenant la requête r_0. Soit $S_{opt} = \{r_{i_0}, ..., r_{i_{k-1}}\}$ une solution optimale (triée par date de fin croissante). Si $r_{i_0} = r_0$, c'est bon ; sinon montrons qu'on peut échanger r_{i_0} avec r_0.

FIGURE 12.3 – Notations de la preuve du lemme d'échange

Comme S_{opt} est trié, pour $j > 0$ on a $f_{i_0} \leq f_{i_j}$. De plus, comme S_{opt} est une solution, ses intervalles ne s'intersectent pas : donc l'inégalité précédente implique aussi $f_{i_0} \leq d_{i_j}$: les intervalles suivants de S_{opt} débutent après la fin de r_{i_0}. Comme $f_0 \leq r_{i_0}$ (r_0 se termine avant r_{i_0}), on peut bien remplacer r_{i_0} par r_0 sans créer d'intersections.
Le nouvel ensemble obtenu contient toujours autant d'intervalles, il est donc également optimal : il existe une solution optimale contenant r_0 le premier choix glouton.

- Récurrence : On montre par récurrence forte sur le nombre d'évènements que l'algorithme glouton renvoie le bon résultat.
  - Initialisation : Trivial pour n=0.
  - Hérédité : Soit $n > 0$. L'algorithme glouton choisit tout d'abord r_0. Les intervalles suivants qu'il prend ne s'intersectent pas avec r_0 : ce sont des intervalles de $\mathscr{R}' = \{r_j \, t.q. \, r_j \cap r_0 = \emptyset\}$. L'algorithme glouton renvoie donc $\{r_0\} \cup S'$ avec S' une solution de $\mathscr{R}'$; et par hypothèse de récurrence S' est solution optimale de $\mathscr{R}'$.
    Reste à prouver que S est une solution optimale pour $\mathscr{R}$. D'après le lemme d'échange, il existe une solution optimale de la forme $\{r_0\} \cup E'$ avec E' des intervalles. Or, tous les intervalles de E' n'intersectent pas r_0 : ce sont donc des intervalles de $\mathscr{R}'$. En particulier, comme $\mathscr{R}'$ est optimal, $|\mathscr{R}'| \geq |E|$ et donc $|\{r_0\} \cup S'| \geq |\{r_0\} \cup E|$.
    On a montré que la solution gloutonne choisit au moins autant d'intervalles qu'une solution optimale : elle est donc optimale.

Remarque. Une implémentation de cette algorithme glouton se fait en $O(n \log_2 n)$: d'abord on trie les intervalles (en $O(n \log n)$). Ensuite, on prend un intervalle à la condition qu'il n'intersecte pas le dernier intervalle que l'on a pris. Comme les intervalles sont triés par date de fin, cela suffit à garantir que

l'intervalle que l'on veut prendre n'intersecte *aucun* intervalle précédent : le test de non-intersection avec les intervalles déjà pris se fait ainsi en temps constant[3] ! Soit une complexité totale en $O(n \log_2 n)$.

1 Plus d'exemples

1.0 Stockage de fichiers

On dispose de n de poids respectifs $p_0, ..., p_{n-1}$. On veut ranger ces fichiers possibles dans un disque dur de capacité maximale P. L'objectif est de ranger le plus de fichiers possibles[4].

On propose l'algorithme glouton suivant : « prendre en priorité les fichiers de plus faible poids ».

Prouvons-le correct. Quitte à trier, supposons les p_i croissants.

- Lemme d'échange : Soit $S_{opt} = \{p_{i_0}, ..., p_{i_k}\}$ une solution optimale non-vide[5]. On a donc $\sum_{j=0}^{k} p_{i_j} \leq P$. Comme $p_0 \leq p_{i_0}$, cette majoration reste vraie en échangeant p_{i_0} avec p_0. On obtient ainsi une solution optimale qui contient p_0 le premier choix glouton.
- Récurrence : Prouvons par récurrence forte sur le nombre n de fichiers que l'algorithme glouton crée une solution optimale sur une entrée à n fichiers et avec poids maximal P quelconque.
  - Notons que c'est trivial pour tous les cas où $P < p_0$ puisque tout fichier est alors trop lourd et l'algorithme glouton renvoie $\emptyset$ qui est optimal car seule solution possible. On ignore donc ces cas par la suite.
  - Initialisation : Trivial pour $n = 0$.
  - Hérédité : S'il y a $n > 0$ fichiers, alors l'algorithme glouton choisit d'abord p_0 puis un ensemble S' d'autres fichiers. Le poids de ces autres fichiers ne peut dépasser $P - p_0$. Notons F' l'ensemble des fichiers plus lourds que p_0 ; par H.R. S' est une solution optimale de $(F', P - p_0)$.
    Or, par lemme d'échange, il existe une solution optimale de la forme $\{p_0\} \cup E'$. Or, dans cette solution, par définition de p_0 tous les éléments de E' sont plus lourds que p_0 donc sont dans F'. De plus, comme c'est une solution, on a $p_0 + \sum_{p \in E'} p \leq P$ donc les éléments de E' ont un poids total inférieur à $P' - p_0$. Ainsi, E' est une soltuion de l'instance $(F', P - p_0)$.
    Par optimalité de S' dans F', $|S'| \geq |E|$. En particulier, la solution renvoyée par l'algorithme glouton a au moins autant de fichiers que $\{p_0\} \cup E$ donc est optimale.

Remarque.

- L'algorithme ci-dessus est en complexité $O(n \log n)$, parce qu'il faut commencer par trier.
- La comparaison avec le problème du sac à dos est frappante : on ne connait pas de solution polynomiale au sac à dos (et on pense qu'il n'y en a pas car on pense que $P \neq NP$) ; pourtant si on fixe toutes les valeurs du sac à dos à 1 on obtient le problème que l'on vient de résoudre avec un *glouton*.

1.1 Rendu de monnaie

Le problème du **rendu de monnaie** est le suivant : étant donnée une somme s, comme rendre[6] la somme s en utilisant le moins de pièces/billets[7] possibles ?

Exemple. en euros, pour rendre $s = 48$, le nombre minimal de pièces à utiliser est $5 : 48 = 20+20+5+2+1$.

3. En particulier, le tri nous permet aussi de gagner en efficacité sur les tests d'intersection. On dit que c'est un prétraitement qui permet d'accélérer l'algorithme (et de le rendre correct - c'est toujours sympa).
4. Autrement dit, c'est un problème du sac à dos où tous les poids valent 1
5. En toute rigueur, il faudrait traiter à part le cas trivial où toute solution optimale est vide.
6. « Rendre » signifie ici « donner ».
7. Pour simplifier, je parlerai uniquement de pièces dans la suite

1.1.0 En euros

On propose l'algorithme glouton suivant : « Utiliser la plus grosse pièce possible jusqu'à avoir tout rendu ». Remarquez que c'est l'algorithme qui a été utilisé pour rendre la monnaie dans l'exemple précédent.

Montrons qu'il est optimal... mais cette fois-ci ; la preuve ne suivra pas exactement le schéma habituel. Pour simplifier la preuve, je vais me limiter aux pièces suivantes : 1, 2, 5, 10.

Notons que dans toute solution optimale, il y a au plus :

- Une pièce de 5 : si on en utilise deux, on peut les remplcacer par une de 10.
- Deux de 2 : si on en utilise 3, on peut les remplacer par une de 5 et une de 1.
- Une de 1 : idem que 5.

Aussi, dans toute solution optimale, il y a au plus quatre pièces qui ne sont pas des 10. Mais de plus, on n'utilise pas ces quatre pièces à la fois car $5+2+2+1=10$: donc les pièces qui ne sont pas des dix se somment à au plus 9. Donc le nombre de 10 utilisés est $\lfloor \frac{s}{10} \rfloor$, c'est à dire le nombre maximal de fois où l'on peut prendre un 10 : c'est exactement ce que fait l'algorithme glouton.

On conclut ensuite par récurrence : si $s < 10$, on vérifie que l'algorithme glouton est optimal. Sinon, d'après ce qui précède[8], l'algorithme glouton, comme toute solution optimale, rend $\lfloor \frac{s}{10} \rfloor$ pièces de 10 puis rend $s - \lfloor \frac{s}{10} \rfloor$. Or cette seconde étape est faite optimalement d'après l'initialisation, et on conclut.

1.1.1 Dans un système monétaire quelconque

La preuve précédente fonctionnait car on pouvait proposait des remplacements efficaces de plusieurs petites picèes par une grosse : par exemple, trois 2 remplacés par un 5 et un 1. Mais cela n'est pas le cas dans tout système monétaire ! Considérez par exemple un système monétaire qui a des pièces de 6, de 4 et de 1. Alors pour rendre 8, l'algorithme glouton utilise $8 = 6 + 1 + 1$ alors que $8 = 4 + 4$ est mieux !

Autrement dit, dans un système monétaire quelconque l'algorithme glouton ne fonctionne pas. Heureusement, on peut trouver une formule de récurrence : le nombre minimal $n(s)$ de pièces à utiliser pour rendre s peut se définir ainsi :

$$n(s) = \begin{cases} 0 & \text{si } s = 0 \\ 1 + \min\{s - p \text{ où } p \text{ est une pièce tq } p \leq s\} & \text{sinon} \end{cases}$$

Cette formule fonctionne peut importe le système monétaire : on peut donc l'utiliser. Comme elle risque d'appeler plsuieurs fois les mêmes valeurs intermédiaires, on a tout intérêt à utiliser de la programmation dynamique !

Exercice. Quelle complexité obtient-on avec de la programmation dynamique bien écrite ?

1.2 Stay tuned !

Plusieurs algorithmes autres gloutons seront vus en MP2I ou en MPI :

1.2.0 Algorithme de Huffman

L'algorithme de Huffman est un algorithme glouton qui prend en entrée un texte et renvoie un encodage binaire des lettres du texte de sorte que à ce que l'encodage de tout le texte (concaténation des encodages des lettres) soit le plus léger possible (tout en restant décodable).

Nous le verrons le mois prochain, dans le chapitre sur l'algorithmique du texte !

8. Qui ressemble donc beaucoup à un lemme d'échange...

1.2.1 Algorithme de Kruskall

En MPI, vous étuierez le problème de trouver un arbre couvrant de poids minimal : étant donné un graphe non-orienté aux arêtes pondérées, comment trouver un sous-graphe qui est un arbre dont la somme des arêtes est le plus faible possible ? L'algorithme de Kruskall est un glouton qui procède ainsi : « prendre l'arête de poids le plus faible qui ne crée pas de cycle ; répéter jusqu'à avoir pris n-1 arêtes ».

Figure 12.4 – Un arbre couvrant de poids minimal

1.2.2 Algorithme ID3

En MPI, vous étudierez le problème de construire un arbre de décision. C'est un arbre qui permet de classer des données dans différentes catégories en fonction de critères binaires.

Figure 12.5 – Un arbre de décision

L'algorithme ID3 est un algorithme qui essaye de construire un arbre à partir d'un ensemble de données déjà classées, de manière gloutonne : il met en racine la question binaire qui maximise le gain d'information [9], et continue récursivement.

Cependant, cet algorithme glouton-ci n'est pas optimal au sens où il ne crée pas un arbre de décision de hauteur minimale - mais il est rapide et ses arbres ne sont tout de même pas trop mauvais !

1.3 Message d'utilité publique

Les gloutons qui renvoient une solution optimale sont l'exception et non la règle.

Au lieu de chercher un glouton, on peut plutôt chercher une expression d'une solution optimale, souvent récursive [10].

9. Notion compliquée que vous verrez en MPI.
10. Et on peut souvent optimiser l'implémentation de cette écriture récursive par programmation dynamique.

2 (HP) Théorie des matroïdes

Il y a un cas où l'on *sait* que l'algorithme glouton fonctionne : les matroïdes. C'est peut-être un peu compliqué (et tout à fait hors-programme).

2.0 Matroïde

Définition 4 (Matroïde).
Soit E un ensemble fini non-vide et $\mathcal{I} \subseteq \mathcal{P}(E)$ également non-vide. On dit que $\mathcal{I}$ est un **matroïde** de E lorque :

- Échange : Pour tous $X, Y \in \mathcal{I}$, si $|X| < |Y|$ alors il existe $e \in Y \backslash X$ tel que $X \cup \{e\} \in \mathcal{I}$.
- Hérédité : Pour tout $X \in \mathcal{I}$, tout $Y \subseteq X$ vérifie $Y \in \mathcal{I}$

Les éléments de $\mathcal{I}$ sont appelés des **indépendants**.

Remarque. Dans l'idée, les indépendants sont des solutions d'un problème. Les deux conditions se comprennent alors ainsi :

- Échange : si une solution Y est plus grande que X, alors on peut agrandir la solution X en lui un élément bien choisi de Y
- Hérédité : une « sous-solution » est une solution.

Définition 5 (Indépendant maximal).
Soit $\mathcal{I}$ un matroïde de E et $Y \in \mathcal{I}$. On dit que X est **maximal** s'il est maximal pour l'inclusion parmi les indépendants.

Remarque. Autrement dit, un indépendant est maximal si on ne peut plus lui rajouter d'éléments.

Lemme 6 (Indépendants maximaux d'un matroïde).
Tous les indépendants maximaux d'un matroïde ont le même cardinal.

Démonstration. Soit X et Y deux indépendants. Par échange, si $|X| < |Y|$, alors il existe $e \in Y \backslash X$ tel que $\{e\} \cup X$ soit aussi indépendant : en particulier, X n'était pas maximal. En particulier, c'est absurde si X et Y sont maximaux mais de cardinaux distincts. □

Définition 7 (Matroïde pondéré).
Un **matroïde pondéré** $\mathcal{I}$ de E est un matroïde accompagné d'une fonction de poids $w : E \to \mathbb{R}_+$.
On définit le poids de $X \in \mathcal{P}(E)$, noté $w(X)$, comme $\sum_{x \in X} w(x)$

2.1 Algorithme glouton

On s'intéresse au problème de trouver un indépendant de poids maximal dans un matroïde pondéré.

On propose l'idée gloutonne suivant : trier les éléments de $E = \{e_0, ..., e_{n-1}\}$ par poids décroissant. Partir de l'ensemble vide, et pour chaque e_i l'ajouter aux éléments déjà pris si ce que l'on obtient est un indépendant. Autrement dit :

Algorithme 20 : Glouton pour un indépendant de poids maximal

Entrées : $(\mathcal{I}, E, w)$ un matroïde pondéré
Sorties : M un indépendant de poids maximal

```
Trier E = {e_0, ..., e_{n-1}} par poids décroissant.
M ← ∅
pour i de 0 à n − 1 faire
    si A ∪ {e_i} ∈ I alors
        A ← A ∪ {e_i}
renvoyer A
```

Théorème 8 ((HP) Optimalité de glouton dans un matroïde pondéré).
L'algorithme précédent renvoie un indépendant de poids maximal.

Démonstration.

- Lemme d'échange : Notons e_k le premier élément choisi par glouton, autrement dit le premier élément ajouté à A. Soit S_{opt} un indépendant de poids maximal. Si $e_k \in S_{opt}$, on a prouvé le lemme d'échange. Sinon, par propriété d'échange des matroïdes, on peut ajouter à $\{e_k\}$ un élément x de S_{opt}. En réappliquant l'échange à $\{e_A, x\}$ et S_{opt}, on peut en ajouter un second : etc etc, par $|S_{opt}| - 1$ applications de la propriété d'échange on construit X un indépendant qui qui contient e_A et $|S_{opt}-1$ éléments de S_{opt}. Notons e_S l'élément de S_{opt} qui n'est pas dans X.
  Come $\{e_S\} \subseteq S_{opt}$, $\{e_S\}$ est un indépendant : en particulier, comme e_A est le premier élément de E dont le singleton forme un indépendant on a $w(e_A) \geq w(e_S)$. Mais on peut alors conclure :

$$\begin{aligned} w(X) - w(S_{opt}) &= \sum_{x \in X} w(x) - \sum_{y \in S_{opt}} w(y) \\ &= w(e_A) - w(e_S) && \text{car ce sont les seuls termes différents de } X \text{ et } S_{opt} \\ &\geq 0 && \text{d'après ce qui précède} \end{aligned}$$

  Ainsi, X est un indépendant de poids supérieur à S_{opt} (qui est de poids maximal), donc est de poids maximal.

- Récurrence :
  - Immédiat si $|E| = 0$
  - Sinon, en notant e_k le premier choix glouton, on pose $E' = \{e_0, ..., e_{k-1}\}$ et $\mathcal{I}' = \{X \subseteq E' \; tq \; X \cup \{e_k\} \in \mathcal{I}\}$. Par propriété d'hérédité, tous les indépendants de $\mathcal{I}'$ sont des indépendants de $\mathcal{I}$ et on en déduit que $\mathcal{I}'$ est un indépendant de $\mathcal{E}'$: l'algorithme glouton renvoie donc une solution optimale sur $\mathcal{I}'$.
    On conclut alors à l'aide du lemme d'échange comme dans les preuves gloutonnes précédentes.

□

Remarque.

- Les matroïdes sont hors-programme, mais ils permettent de comprendre pourquoi « lemme d'échange + récurrence » fonctionne souvent pour les gloutons : c'est parce que beaucoup de problèmes qui ont une solution gloutonne sont des matroïdes cachés.
- Attention, tous les problèmes admettant une solution gloutonne ne sont pas des matroïdes ! Premièrement parce que « algorithme glouton » est défini assez vaguement, et deuxièmement parce que l'informatique n'est pas triviale [11].

11. Ce serait trop beau d'avoir une méthode magique pour savoir si un problème admet ou non une solution gloutonne...

Exercice. Soit $G = (S, A)$ un graphe non-orienté pondéré par $w : A \to \mathbb{R}_+$. On dit que $X \in A$ est une forêt si le graphe obtenu en ne gardant que les arêtes de A est une forêt. Alors l'ensemble des forêts est un matroïde pondéré de A. En particulier, en choisissant la bonne relation d'ordre, on en déduit un algortihme pour trouver une forêt de poids minimal : c'est l'algorithme de Kruskall (MPI) !