Langages formels (11X003) - Automne 2025

1. Algorithmes gloutons (I)

Enseignant: Arnaud Casteigts (et F. Raynaud) *Assistant: Brian Pulfer*
Moniteurs: T. von Düring & A. Maendly

Ces notes correspondent au premier cours sur le sujet.

2.1 Algorithme glouton

Un algorithme glouton est un algorithme qui effectue, à chaque étape, le meilleur choix possible sur le moment, sans retour en arrière ni anticipation des étapes suivantes. On parle aussi de choix localement optimal. Pour certains problèmes, cette stratégie fonctionne très bien. Pour d'autres, elle fonctionne moins bien (et pour certaines, pas du tout). Mais dans tous les cas, elle garantit un temps d'exécution rapide.

Les algorithmes gloutons s'adressent généralement aux problèmes d'optimisation, où l'on cherche à minimiser ou maximiser un critère (p.ex. la taille ou le coût) parmi un ensemble de solutions possibles. Nous allons voir quelques exemples. Pour chaque problème, nous appelons OPT la valeur de la solution optimale, à laquelle nous pouvons comparer la solution trouvée.

2.2 Exemples

2.2.1 Voyageur de commerce

Voyageur de commerce (TSP, pour traveling salesperson problem en anglais) : étant donné un ensemble de n villes, une ville de départ et un coût entre certaines paires de villes (concrètement, étant donné un graphe pondéré), l'objectif est de trouver une tournée qui passe une fois par chaque ville et termine au point de départ, en minimisant la somme des coûts. Il s'agit d'un problème d'optimisation, qui s'avère être NP-difficile. Le problème de décision associé ("existe-t-il une tournée de coût inférieur à k ?") est NP-complet. Nous allons nous concentrer sur la version d'optimisation.

Solution naive qui teste toutes les solutions ? Complexité factorielle. Pourquoi ?

Algorithme glouton : à chaque étape, choisir la ville non-visitée dont le coût depuis la ville actuelle est le plus petit (l'algorithme "Nearest neighbor").

Complexité de cet algorithme ? Clairement faisable en $O(n^2)$: à chaque étape, on cherche le minimum parmi les villes restantes (p.ex. en parcourant une liste). On regarde donc n fois

chacune des n villes dans le pire des cas (en supposant que le calcul d'une distance entre deux villes prend un temps constant).

Qualité de la solution ? Clairement pas optimale (ex : villes sur les sommets d'un parallélogramme). En fait, cet algorithme donne une approximation qui peut être arbitrairement mauvaise : pour toute constante α, il existe des instances où la qualité de la solution est pire que $\alpha \cdot OPT$.

2.2.2 Rendu de monnaie

Problème du rendu de monnaie (change making, en anglais) : Étant donné n pièces de monnaies, chacune d'une valeur donnée, et une somme désirée, l'objectif est d'atteindre cette somme en utilisant le moins de pièces possibles.

Une instance du problème peut être représentée comme un multi-ensemble de n entiers représentant la valeur des pièces (potentiellement avec des répétitions s'il y a plusieurs pièces de même valeur) et un entier cible (la somme à atteindre).

Un algorithme glouton correspondrait ici à sélectionner de manière répétée la plus grande pièce possible sans dépasser la somme voulue.

Exemple 1 : Pièces {2 francs, 2 francs, 1 franc, 50 centimes, 20 centimes, 20 centimes, 10 centimes}, autrement dit $\{200, 200, 100, 50, 20, 20, 10\}$ et objectif 380. L'algorithme glouton fonctionne bien sur cet exemple et va trouver une solution à 5 pièces, $\{200, 100, 50, 20, 10\}$, ce qui est en effet optimal.

Exemple 2 : Pièces $\{25, 20, 20, 10, 5\}$ (ancien système indien) et objectif 40. L'algorithme glouton va trouver une solution à trois pièces, $\{25, 10, 5\}$. C'est bien, mais il existait une solution à deux pièces : $\{20, 20\}$ qu'une approache gloutonne va rater.

Exemple 3 : Pièces $\{25, 20, 20, 5\}$ et objectif 40. Ici, l'algorithme va échouer. Il ne fallait pas choisir 25 du tout ! L'approche gloutonne échoue donc parfois.

Morale de l'histoire : les choix effectués par l'algorithme à un instant donné peuvent avoir une forte influence sur le futur. L'algorithme glouton fonctionne parfois très bien, mais parfois pas. Savoir identifier les problèmes, ou les variantes de problèmes, pour lesquels il fonctionne bien est un atout. (En l'occurence, il est optimal pour les systèmes de monnaies dits canoniques, par exemple les systèmes CHF ou EUR avec un nombre illimité de pièces.)

2.2.3 Sac à dos

Le problème du sac à dos (knapsack, en anglais) est plus général que celui du rendu de monnaie. Ici, une instance correspond à un ensemble de n objets qui ont chacun une valeur et un poids. L'objectif est de trouver un sous-ensemble d'objets de valeur maximum sans

dépasser un poids total donné.

Une instance se présente comme $(\{(v_1, w_1), (v_2, w_2), \ldots, (v_k, w_k)\}, T)$, où (v_i, w_i) correspond à la valeur et au poids (weight) de l'objet i, et T est le poids total à ne pas dépasser. Par exemple $(\{(100, 20), (120, 30), (10, 10), (60, 10)\}, 50)$.

Approche naïve : essayer tous les sous-ensembles possibles. Quelle est la complexité ? Dans l'exemple précédent, c'est faisable, mais s'il y a n objets, cela fait 2^n sous-ensembles à tester, ce qui est exponentiel... (pas terrible).

Que pourrait être une approche gloutonne à ce problème ? Il y a plusieurs possibilités. L'une des plus naturelles consiste à choisir, à chaque étape, l'objet qui maximise le rapport valeur/poids (la densité de l'objet) sans excéder la limite de poids. Pour faire cela, on peut commencer par trier les objets par densité décroissante, à savoir dans notre exemple :

$((60, 10), (100, 20), (120, 30), (10, 10))$. (les densités respectives sont $6, 5, 4, 1$)

On parcourt ensuite les éléments un à un, dans cet ordre, en les ajoutant à notre solution tant que cela ne dépasse pas le seuil (ici 50). On ajoute donc $(60, 10)$, puis $(100, 20)$, puis on ne peut plus ajouter $(120, 30)$, on continue donc en ajoutant $(10, 10)$. On obtient une valeur totale de 170. Pouvait-on faire mieux ? Oui : en renonçant au meilleur élément ! En effet, la solution $\{(100, 20), (120, 30)\}$ atteint une valeur de 220 sans dépasser le seuil.

L'algorithme glouton rate donc encore l'optimum. Cependant, on peut le modifier très légèrement de sorte à garantir une 2-approximation, c'est à dire une solution qui s'approche à coup sûr à un facteur 2 de l'optimum (en l'occurence, une valeur totale d'au moins OPT/2).

Voici l'algorithme :

1. Trier les objets par densité décroissante (comme précédemment)
2. Prendre les objets dans l'ordre et s'arrêter dès qu'un objet i qui ne peut pas être ajouté
3. Comparer la valeur totale des objets pris jusqu'à présent à la valeur de l'objet i
4. Si i a une plus grande valeur, renvoyer i, sinon renvoyer les objets sélectionnés

Pourquoi cela donne une 2-approximation ? Appelons S l'ensemble des objets sélectionnés. Imaginez d'abord que l'on puisse couper l'objet i pour arriver exactement à la capacité du sac à dos. On aurait alors clairement une solution optimale, car les objets sont triés par densité décroissante. Cela implique que la valeur optimale OPT est inférieure à $v(A) + v_j$. Puisqu'il y a deux termes dans cette addition, le plus grand des deux est au moins $OPT/2$, cet algorithme est donc bien une 2-approximation.