Algorithmique (12X001) - Automne 2023

6. Algorithmes gloutons

Enseignant: Arnaud Casteigts *Assistant: Brian Pulfer*

6.1 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 est très rapide.

Les algorithmes gloutons s'adressent généralement aux problèmes d'optimisation, où l'on cherche à minimiser ou maximiser un objectif parmi toutes les solutions possibles. Nous allons voir quelques exemples. Dans chacun des cas, nous appelerons OPT la valeur de la solution optimale, à laquelle nous comparerons la solution trouvée. En ce qui concerne la complexité algorithmique, nous nous intéresserons surtout à la complexité en temps de calcul.

6.2 Exemples

6.2.1 Voyageur de commerce

Voyageur de commerce (TSP, pour *traveling salesperson problem* en anglais) : étant donné une ville de départ, un ensemble de n villes à visiter et un coût entre chaque paire de villes, trouver un circuit qui passe une fois par chaque ville et retourne au point de départ, en minimisant la somme des coûts.

Complexité du problème ? NP-complet pour la variante décisionnelle (existe-t-il un circuit de coût inférieur à k ?). NP-difficile pour la variante optimisation (trouver le plus petit).

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 minimum (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 : il a été montré que pour toute constante α, il existe des instances où la qualité de la solution est pire que $\alpha \cdot OPT$.

6.2.2 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 objectif, 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).

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 greedy va trouver une solution à trois pièces, $\{25, 10, 5\}$, alors qu'une solution à deux pièces existait, $\{20, 20\}$.

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

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.)

6.2.3 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 de valeur maximum sans dépasser un poids total donné.

Une instance peut être donné comme un ensemble de couples $(valeurs, poids)$, par exemple $\{(100, 20), (120, 30), (10, 10), (60, 10)\}$ ainsi que le poids total à ne pas dépasser, p. ex. 50.

Approche naïve : essayer tous les sous-ensembles possibles. Quelle est la complexité ?

Dans l'exemple précédent, c'est faisable, mais en général, c'est exponentiel, car un ensemble de n éléments a 2^n sous-ensembles possibles... pas terrible.

Que pourrait être une approche gloutonne à ce problème ? Il y a plusieurs possibilités, que vous verrez lors des séances d'exercices. L'une des plus naturelles consiste à choisir, à chaque étape, l'objet qui maximise le rapport valeur/poids sans excéder la limite de poids. Pour faire cela, on peut commencer par trier les couples par ratio valeur/poids décroissants, ce qui dans notre exemple donne la liste suivante :

$((60, 10), (100, 20), (120, 30), (10, 10))$.

On parcourt ensuite les éléments un à un, dans cet ordre, en les ajoutant à notre solution si 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'approche gloutonne rate donc encore l'optimum, mais cette fois-ci, on peut la modifier très légèrement pour obtenir 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, au moins OPT/2).

Commençons par nettoyer l'instance de départ pour enlever tous les objets qui dépassent à eux seuls le seuil autorisé (ces objets ne servent à rien). On applique ensuite l'algorithme glouton comme indiqué précédemment, mais cette fois-ci on s'arrête dès qu'un objet x dépasse le seuil. À ce moment là, notre solution gloutonne cumule une certaine valeur V et l'objet refusé a lui-même une valeur V_x. Si $V_x > V$, on renvoie l'objet x comme solution (tout seul), sinon, on renvoit la solution gloutonne. Il se trouve que cela est nécessairement supérieur ou égal à OPT/2. Pour montrer cela, il suffit de montrer que OPT est forcément inférieur à $V + V_x$ (laissé en exercice).

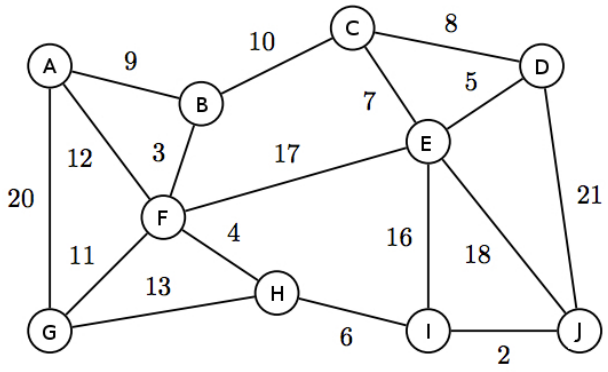
6.3 Quand l'approche gloutonne est optimale

Nous avons vu des exemples ou l'algorithme glouton fonctionne avec plus ou moins de succès. Mais dans certains cas, il trouve toujours l'optimum ! C'est le cas de plusieurs problèmes importants.

6.3.1 Arbre couvrant de poids minimum

Le problème de l'arbre couvrant de poids minimum (MST, pour *minimum spanning tree*, en anglais) est très utilisé, notamment pour faire du routage dans les réseaux. Étant donné un graphe non-orienté, mais pondéré (c.à.d. dont les arêtes ont un coût) à n sommets et m arêtes, l'objectif est de trouver un arbre couvrant de ce graphe (c.à.d. un sous-graphe sans cycle qui connecte tous les sommets) dont la somme des poids est la plus petite possible.

Voici un exemple d'instance pour ce problème :



Il existe deux algorithmes gloutons bien connus pour ce problème : l'algorithme de Kruskal et l'algorithme de Prim. Les deux algorithmes sont faciles à décrire. En revanche, le fait qu'ils trouvent l'optimum systématiquement n'est pas immédiat (mais c'est bien le cas).

Algorithme de Kruskal L'algorithme est conceptuellement très simple :

1. Démarrer avec un arbre vide,
2. Ajouter à l'arbre la plus petite arête qui ne crée pas de cycle,
3. Recommencer l'étape 2 jusqu'à ce que l'arbre ait $n - 1$ arêtes.

Notez qu'en cours d'exécution, la solution partiellement calculée jusqu'à présent n'est pas encore un arbre, puisqu'elle peut être composée de plusieurs fragments non-connexes. On parle alors de forêt. Mais la solution finale est bien un arbre.

À première vue, il peut sembler surprenant que cet algorithme trouve toujours l'optimum. Mais c'est bien le cas. Nous en toucherons deux mots à la fin de ces notes.

Pour implémenter concrètement cet algorithme, on peut commencer par trier les arêtes par ordre croissant, ce calcul a une complexité en $O(m \log m)$ ($= O(m \log n)$, pourquoi ?). Puis les parcourir une seule fois. La difficulté principale est de détecter si l'ajout d'une arête candidate crée un cycle dans l'arbre (afin de la rejeter). On peut faire cela en maintenant à jour une liste des composantes connexes de l'arbre, ce qui est un peu plus coûteux, mais également faisable en temps $O(m \log n)$ (en utilisant une structure de donnée `union-find`). L'algorithme coûte donc $O(m \log n) + O(m \log n) = O(m \log n)$ (pourquoi ?).

Algorithme de Prim Voici l'algorithme :

1. Choisir un sommet de départ (arbitrairement) et le marquer comme appartenant à l'arbre,

2. Trouver la plus petite arête dont une extrémité est marquée et l'autre non,
3. Ajouter cette arête et marquer le sommet correspondant,
4. Recommencer les étapes 2 et 3 jusqu'à ce que tous les sommets soient marqués.

Cet algorithme est similaire à celui de Kruskal, mais il a une propriété supplémentaire très pratique : à tout moment, la solution partiellement calculée est connexe et nous n'avons pas besoin de tester les cycles. Il est donc plus facile à implémenter. Par ailleurs, sa complexité est très légèrement meilleure : $O(m + n \log n)$ (ici, il faut lire le + comme un max) si l'on utilise des tas de Fibonacci pour trouver l'arête la plus petite à chaque étape.

6.3.2 Autres exemples

- L'algorithme de Dijkstra pour la recherche de plus court chemin.
- Algorithme A*
- Algorithme de Huffman pour la compression
- ...

6.3.3 Pourquoi ça marche ?

L'algorithme glouton est optimal pour certains problèmes et moins bon pour d'autres. Nous avons aujourd'hui une bonne compréhension des raisons. Cela a trait aux propriétés du problème lui-même, et en l'occurrence, au fait que la structure recherchée vérifie des axiomes de matroïdes (ou plus fin, de greedoïdes). Ces aspects sont au delà d'un cours de bachelor (nous les verrons peut-être en master). Intuitivement, ces axiomes requièrent que toute solution optimale partielle puisse toujours être augmentée par un élément appartenant à une solution optimale globale (comprendre : il n'y a pas d'impasse dans le calcul, si vous prenez des décisions localement optimales, vous atteindrez un optimum global).