Université de Genève
-
Sciences Informatiques

Algorithmique - TP 02

Noah Munz (19-815-489)

Octobre 2022

Contents

TP 02 – Greedy & Complexité

1 Knapsack

Pour chaque implémentation greedy du knapsack indiquer sa complexité en temps et sa preuve/contre-exemple d'optimalité.

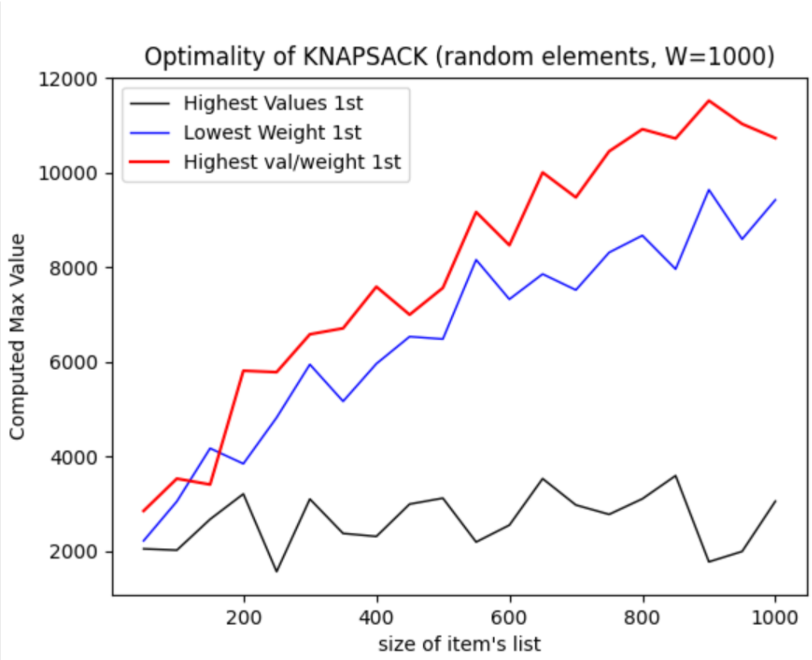
1.1 optimalité

Les 3 implémentations doivent maximiser les caractéristiques respectives suivantes :

(a) La valeur la plus élevée

(b) Le poids le plus bas

(c) Le ratio $\frac{valeur}{poids}$ le plus élevé.



Le graph ci-dessus montre la performance des 3 implémentations demandées sur des listes de paires $(value,\ weight)$ aléatoires de taille variants entre 50 et 1000, pour un poids maximal (W) de 1000.

Comme on peut le voir, l'implémentation (a) n'est pas optimale, mais reste plutôt efficiente comparé à celle en (b) qui reste globalement tout autant catastrophique peut importe la taille, n, de la liste d'items. Ce qui est assez étrange lorsque l'on multiplie le poids maximal autorisé par jusqu'à un facteur 5 vers la fin.

Enfin il paraît assez évident que c'est bien l'implémentation (c) qui est optimale. En effet, pondérer la valeur de chaque objet par sa masse pour obtenir le ratio $\frac{valeur}{poids}$ semble être la chose la plus logique à faire car seul ce ratio prend en compte les 2 données, (poids et valeur) du problème, pour chaque objet.

1.2 complexité

Au niveau complexité, les 3 algorithms sont les mêmes, la seule chose qui chose est la manière dont on trie la liste d'items au début, mais là encore la taille de cette dernière ne varie pas entre (a), (b) ou (c).

Pour chaque algorithme, une implémentation récursive a été choisie. (voir ligne 19-39 de *tp1.py*)

Etant donnée que chaque élément de `items` est visité au maximum 1 fois, on en conclus que le nombre d'appels à `try_rec` et d'itérations de la boucle `while` est au total `n`.
On a donc que la complexité des lignes 29-39 est en $O(n)$ et avant ça, la seule "vraie" opération est le tri (appel à `sorted`) qui s'effectue en $O(n \log n)$.

Le tout s'effectue donc en $O(n) + O(n \log n) = O(n \log n)$.

2 Rendu de monnaie britannique

2.1 Pseudo-code greedy

```
coin(M: money, (c0, c1, ..., cn-1): coin set where c0 > c1 > ... > cn-1) =

    coin_rec(L, i, acc) =
        if L - ci >= 0
            then coin_rec(L - ci, i, acc + [ci])
        else if (i+1 < n && L > 0)
            coin_rec(L, i+i, acc)

    coin_rec(M, 0, NIL)
```

2.2 Pourquoi est-il greedy?

Car il essai de toujours la meilleure option / le meilleur choix localement, pour espérer avoir le meilleur choix / meilleur solution globablement.

2.3 Complexité

Soit S une solution du "coin algorithm" et $d_i := |S|_{d_i}$ le nombre de fois que la pièce c_i est utilisé. (Dans l'exemple en (2.4), on a: $d_0 = 1,\ d_1 = 0,\ d_2 = 6$.)
On a donc que chaque appel récursif pour le même i est fait d_i fois. (Parce que l'on met d_i fois la pièce c_i dans S.) L'algorithme est donc en $O(\sum_{i=0}^{n-1} d_i)$.

On a que $L_{i+1} = L_i \mod c_i\ \ (L_0 = M)$, i.e. $L_3 = (((L_0 \mod c_0) \mod c_1) \mod c_2)$
Soit a div $b := \lfloor \frac{a}{b} \rfloor$, on voit qu'on a également $d_i = L_i$ div c_i, où $0 \leq L_i < c_{i-1}$. En effet, $(a \mod b) < b$.

Ce qui nous permet d'obtenir une borne supérieure sur les d_i : $d_i = L_i$ div $c_i \leq L_i < c_{i-1}$ (si jamais $c_i = 1$). La complexité en temps est donc borné par $\sum_{i=0}^{n-1} c_i$, chaque constante c_i est borné par c_0 (toujours une constante). Le tout est donc borné par $\sum_{i=0}^{n-1} c_0 = n \cdot c_0$, soit $O(n \cdot c_0) = O(n)$.

2.4 Optimalité

Cet algorithme est-il optimal ? Si oui prouvez-le, sinon donnez un contre-exemple.
Non il existe des valeurs de M et des coins sets pour lesquels l'algorithme ne donne pas la solution optiomale.

E.g. M (money) $= 0.31,\ \ C = \{25,\ 10,\ 1\}$

- Greedy : $0.31 \; = \; 1 \cdot 25 + 6 \cdot 1$ (7 pièces)
- Optimal: $0.31 \; = \; 3 \cdot 10 + 1 \cdot 1$ (4 pièces)

3 Minimum Spanning Tree (MST)

3.1 Optimalité

Cet algorithme va-t-il toujours trouver un MST ? Si oui prouvez-le, sinon donnez un contre-exemple.

Soit $G = (V, E)$, le graph t.q. $|E| = n$ et T un MST. L'existance de T est garantie par la connexité du graph.
Soit maintenant F_n le sous-graph à n edges généré par l'algorithme. et $P(n) := \exists T,\; F_n \subseteq T$
i.e. il existe un MST qui contient le sous-graph généré à l'étape n. On montre par récurrence sur n, que $\forall n, P(n)$.

$P(0)$ est vrai, car $F_0 = \varnothing$ et tout ensemble (T y compris) contient l'ensemble vide.

Supposons que $P(k)$ pour un $k \in [\![1,\; n-1]\!]$ et montrons $P(k+1)$.
F_k est contenu dans T donc la somme totale des noeuds dans F_k est déjà "optimale pour F_k" (il n'existe pas d'autre sous graph de taille k de poids strictement plus petit). Donc F_{k+1} est aussi contenu dans T. En effet, soit e l'edge qui a été rajouté pour F_{k+1}, l'algorithme nous garantie qu'il n'existe pas d'autre e (qui ne formerait pas de cycle) de poids plus petit (car la liste est triée) et il nous assure aussi rajouter e à F_k ne provoquera pas de cycle. Pas de cycle (arbre) + poids minimum $\implies F_{k+1}$ appartient à un arbre couvrant.

Par récurrence, on a donc que $P(n)\ \forall n \geq 0$, or un sous-graph de taille $|E| = n$ est un arbre couvrant, donc F_n est un MST. (En effet $P(n)$ devient "il existe un $T = F_n$" car $|T| = n,\ \forall T$)
L'algorithme génère donc toujours un MST.

$\square$

4 4-SAT

4.1 Réduction

Prouvez que 4-SAT est NP-Complet.

On montre que 4-SAT est NP-Complet en prouvant (a) qu'il est dans NP et (b) qu'il est NP-Hard.

(a) On prouve qu'il est dans NP en implémentant une machine de Turing non-déterministe qui résout 4-SAT en un temps polynomial.

Une instance de 4-SAT est une proposition logique C, composée d'un nombre fini de clause c_i, telle que:

- $\forall i :\ c_i = x_{i1} \vee x_{i2} \vee x_{i3} \vee x_{i4}$ (c_i contient 4 éléments)
- $\bigwedge_i c_i = C$

Donc, s'il existe une solution (n quadruplets c_i, t.q. $C \equiv Vrai$), on la choisit par non-determinisme. ($O(n)$)
Puis on vérifie si cette solution est bien correcte, i.e. on vérifie que $\forall i, c_i \equiv Vrai$ ($O(n)$). Si oui on accepte l'entrée sinon on refuse.

Le tout est en $O(n) + O(n) = O(n)$ (polynomial) et détermine bien le problème pour chaque entrée.
Donc 4-SAT $\in$ NP

(b) On prouve que 4-SAT est NP-Hard en effectuant une réduction depuis 3-SAT (3-SAT $\propto$ 4-SAT):
Pour transformer une instance de 3-SAT en instance de 4-SAT on procède comme suit:
Pour chacun des n triplets, c_i, en entrée, on choisit une de ses 3 variables, appelons la y_i.

Sans pertes de généralités, mettons que $y_i := x_{i3}$. On remplace donc chaque bloc par un qui contient y_i et un qui contient $!y_i$. On obtient ainsi $2n$ quadruplets.
i.e. on remplace chaque c_i par $(c_i \vee y_i) \wedge (c_i \vee !y_i)$, ce qui nous donne finalement une formule du style:

$$(x_{11} \vee x_{12} \vee x_{13} \vee y_1) \wedge (x_{11} \vee x_{12} \vee x_{13} \vee !y_1) \ \wedge \ (x_{21} \vee x_{22} \vee x_{23} \vee y_2) \wedge (x_{21} \vee x_{22} \vee x_{23} \vee !y_2) \ \wedge \ ...$$

$$... \ \wedge \ (x_{n1} \vee x_{n2} \vee x_{n3} \vee y_n) \wedge (x_{n1} \vee x_{n2} \vee x_{n3} \vee !y_n)$$

qui est bien une instance de 4-SAT, où l'on a créé $2n$ blocs. $O(n)$.
Donc 4-SAT est NP-Hard et donc 4-SAT est NP-Complet. $\square$