Université de Lille
FST- Département Informatique

ACT-Résumé de Cours

2018–2019
Master d'Informatique-S1
S.Tison

Algorithmes gloutons

Idée intuitive

Le principe d'une méthode gloutonne est très simple : on construit une solution *incrémentalement* en rajoutant à chaque pas un élément selon un critère glouton, i.e. celui qui nous paraît "localement" le meilleur. On fait donc des choix à "court terme" sans jamais les remettre en cause. La construction de la solution est donc souvent très simple, le problème étant plutôt de justifier que la solution construite ainsi est bien optimale ! Quand cette vison à court terme nous donne toujours une solution optimale, on parle d'algorithme glouton exact sinon on parle d'*heuristique gloutonne* -nous y reviendrons plus tard-.

Si on voit la recherche d'une solution comme une exploration de l'arbre des choix, dans le cas d'un algorithme glouton, on construit uniquement et directement -sans backtracking- une -et une seule- branche de l'arbre. Si elle correspond à une solution optimale, on a bien un algorithme glouton exact.

Le schéma général

Le problème

On est souvent dans le cadre de problèmes d'optimisation. Plus précisément, on est le plus souvent dans le cas suivant :

. On a un ensemble fini d'éléments, E.

. Une solution au problème est construite à partir des éléments de E : c'est par exemple une partie de E ou un multi-ensemble d'éléments de E ou une suite (finie) d'éléments de E ou une permutation de E qui satisfait une certaine contrainte.

. A chaque solution S est associée une fonction objectif $v(S)$: on cherche donc une solution qui maximise (ou minimise) cette fonction objectif.

Le schéma de la méthode gloutonne

Il est basé sur un critère **local** de sélection des éléments de E pour construire une solution optimale. En fait, on travaille sur l'objet " solution partielle"- "début de solution"- et on doit disposer de :

. `select` : qui choisit le meilleur élément restant selon le critère glouton.

. `complete?` qui teste si une solution partielle est une solution (complète).

. `ajoutPossible?` qui teste si un élément peut être ajouté à une solution partielle, i.e. si la solution partielle reste un début de solution possible après l'ajout de l'élément. Dans certains cas, c'est toujours vrai !

.`ajout` qui permet d'ajouter un élément à une solution si c'est possible.

L'algorithme est alors :

```
SCHÉMA d'ALGO GLOUTON
// on initialise l'ensemble des "briques"
// élémentaires des solutions.
Ens.init() ;
// on initialise la solution :
// ensemble (ou suite) "vide" ou..
Sol.Init() ;
while (Non Sol.complete ?() et
 Ens.NonVide ?()) do
    //on choisit x selon critère glouton
    x ← Ens.select();
    if Sol.ajoutPossible(x) then
      Sol.ajout(x) ; fsi ;
    //dans certains problèmes, toujours le cas
    if CertainesConditions then
      Ens.retirer(x) ;
    // selon les cas, x considéré une fois ou plus
end
// la Solution partielle est a priori complète
 return Sol ;
```

Pour sélectionner, on trie souvent tout simplement la liste des éléments selon le critère glouton au départ ; on balaye ensuite cette liste dans l'ordre.

Ceci est un schéma général qui a l'avantage et les inconvénients d'un schéma : dans certains cas, c'est encore plus simple ! par exemple, lorsque la solution recherchée est une permutation, en général l'algorithme se réduit au tri selon le critère glouton ! dans d'autres cas, les "solutions" sont un peu plus compliquées et on a besoin d'un schéma un peu plus sophistiqué.

Complexité

Soit n le cardinal de E. La complexité est donc souvent de l'ordre de $n\ logn$ (le tri selon le critère) $+n * f(n)$, si $f(n)$ est le coût de la vérification de la contrainte et de l'ajout d'un élément : les algorithmes gloutons sont en général efficaces.

Correction

Encore faut-il que l'algorithme donne bien une solution optimale et qu'on sache le prouver... : là réside souvent la difficulté dans les algorithmes gloutons.

Les preuves de correction d'algorithmes gloutons, sont souvent basées sur une propriété appelée " Propriété d'échange" :

*P*ropriété d'échange : Soit une solution quelconque différente de la gloutonne : on peut la transformer en une autre solution au moins aussi bonne et "plus proche" de la solution gloutonne .

On peut donc ainsi montrer par contradiction qu'il ne peut y avoir de solution strictement meilleure que la solution gloutonne : supposons la gloutonne non optimale ; soit une solution optimale la plus proche possible de la solution gloutonne -si l'ensemble des solutions est fini, il y en a (au moins) une-. Par la propriété d'échange, on transformera cette solution en une aussi bonne -donc toujours optimale- et plus proche de la gloutonne et on aboutira donc à une contradiction et on aura montré que la gloutonne est optimale !

Souvent pour mesurer la "proximité" entre deux solutions, on 'trie" chaque solution (qui est souvent une liste ou un ensemble) selon le critère glouton et on regarde le premier élément qui diffère entre les deux solutions : plus il est loin, plus les solutions sont proches. Vous pouvez imaginer deux personnes construisant simultanément leurs solutions, l'une appliquant la stratégie gloutonne, l'autre non : cela correspond à la première fois où les deux personnes n'ont pas la même stratégie. Plus formellement, on peut souvent définir un ordre total sur les solutions $<$ telle que *Gloutonne* soit maximale (ou minimale) parmi les solutions. Montrer la propriété d'échange correspond ensuite à montrer que si $Gloutonne > S$ alors, il existe une solution S' telle que $Gloutonne \geq S' > S$, avec S' au moins aussi bonne que S. Soit S maximale pour l'ordre $<$ parmi les solutions optimales (elle existe quand l'ensemble des solutions est fini) : d'après ce qui précède, Gloutonne est S et donc est optimale.

Ce schéma de preuve a l'avantage d'être général -i.e. de s'appliquer à la plupart des algorithmes gloutons- mais l'inconvénient d'être souvent lourd ; dans certains cas, il y a une fa çon plus directe -mais moins "générique"- pour prouver la correction.

Formalisation : Il existe une élégante formalisation des algorithmes gloutons qui est basée sur la théorie des **matroïdes** introduits par Whitney en 1935 pour des notions d'indépendance en algèbre linéaire.

Exemples classiques

Voici quelques exemples classiques d'algorithmes gloutons : les algorithmes de Prim et de Kruskal de calcul d'un arbre de recouvrement d'un graphe de poids minimal, l'algorithme des plus courts chemins dans un graphe de Dijkstra, le code de Huffman, de nombreuses versions de problèmes d'affectations de tâches...

Pour mettre au point un algorithme glouton, il faut :

- Trouver un critère de sélection : souvent facile, mais pas toujours
- Montrer que le critère est bon, c.à.d. que la solution obtenue est optimale : nécessite souvent un peu de réflexion !
- L'implémenter : en général facile et efficace.

Arbre des solutions, programmation dynamique et algorithmes gloutons

Dans de nombreux problèmes d'optimisation ou de recherche de stratégie, on peut représenter l'ensemble des solutions sous forme d'un arbre, les solutions pouvant être vues comme une suite de choix : une branche correspond à une stratégie/solution.

Dans le cas de la programmation dynamique, on parcourt toutes les solutions mais on remarque que de nombreux nœuds de l'arbre correspondent aux mêmes sous-problèmes : l'arbre peut donc être représenté de façon beaucoup plus compacte comme un graphe acyclique (DAG : directed acyclic graph). C'est le rôle de la table utilisée traditionnellement en programmation dynamique. Donc on parcourt efficacement l'espace de toutes les solutions car on partage les sous-problèmes communs.
Dans le cas des algorithmes gloutons, le critère glouton permet de choisir la branche de l'arbre sans avoir à explorer tout l'arbre.

Trois exemples de preuve

Un raisonnement type

Le problème : Sont données n demandes de réservation -pour une salle, un équipement ...- avec pour chacune d'entre elles l'heure de début et de fin (on supposera qu'on n'a pas besoin de période de battement entre deux réservations). Bien sûr, à un instant donné, l'équipement ne peut être réservé qu'une fois ! On cherche à donner satisfaction au maximum de demandes.

Donc, le problème est défini par :

Donnée :
n –le nombre de réservations
$(d_1, f_1), ..., (d_n, f_n)$ –pour chacune d'entre elles, le début et la fin :

Sortie : les réservations retenues i.e. $J \subset [1..n]$ tel que :
. elles sont compatibles : si $i \in J, j \in J, i \neq j$, alors $d_i >= f_j$ ou $f_i <= d_j$
. on a satisfait le maximum de demandes, i.e. $|J|$ (le cardinal de J) est maximal.

L'algorithme glouton sera par exemple :

Input: n, le nombre de réservations
$(d_1, f_1), ..., (d_n, f_n)$ leurs débuts, fins
Output: une liste optimale de demandes acceptées
// date de disponibilité de la ressource
int $lib \leftarrow 0$;
//nb de résa prises en compte
int $nb \leftarrow 0$;
//liste des réservations acceptées
Liste g ;
g.Init() ; // g=Vide ;
Trier les demandes dans *l'ordre croissant de fin* ;
for *chaque demande dem* **do**
  if *(dem.debut* $\geq$ *lib)* **then**
    g.ajouter(dem) ;
    //dem acceptée
    $lib \leftarrow dem.fin$;
    //ressource bloquee jusque dem.fin
    $nb++$;
end

Comment prouver que l'algorithme est correct ?

— On montre facilement que la solution est correcte, i.e. les réservations retenues sont bien compatibles ;
Pour cela on peut par exemple prendre pour invariant :
{*Les demandes de g sont compatibles et finissent toutes au plus tard à Lib*}.

— Le fait que la solution soit optimale est un peu plus difficile à montrer.

Préliminaire : on définit une notion de distance sur les solutions. Soient deux solutions A, B différentes ; soit i le plus petit indice (les solutions étant supposées triées dans l'ordre des fins croissantes), tel que A_i soit différent de B_i ou tel que A_i soit défini et pas B_i ou le contraire.
On notera $dist(A, B) = 2^{-i}$

Preuve de l'optimalité de l'algorithme :
Raisonnons par l'absurde : Supposons que la solution gloutonne g ne soit pas optimale. Soit alors o une solution optimale telle que $dist(g, o)$ soit minimale (une telle o existe bien, l'ensemble des solutions étant fini). Soit donc i telle que $dist(g, o) = 2^{-i}$; il y a trois possibilités :

cas 1 : $g_i \neq o_i$: alors, la demande o_i a une date de fin au moins égale à celle de g_i : sinon comme elle est compatible avec les précédentes, l'algorithme glouton l'aurait examinée avant g_i et l'aurait choisie.
Mais alors, si on transforme o en o' en remplaçant l'ctivité o_i par g_i, on obtient bien une solution, de même cardinal que o donc optimale, et telle que $dist(o', g) < dist(o, g)$: on aboutit à une contradicition !
Note : on utilise ici ce qu'on appelle *la propriété d'échange*.

cas 2 : o_i n'est pas définie : alors $|o| < |g|$: contradiction, o ne serait pas optimale.

cas 3 : g_i n'est pas définie : impossible car l'activité o_i étant compatible avec les précédentes, l'algo glouton l'aurait choisie.

Donc, dans tous les cas, on aboutit à une contradiction ; l'hypothèse "la solution gloutonne g n'est pas optimale" est fausse : la solution gloutonne est bien optimale !

Une autre version de la preuve

On peut faire le même raisonnement en utilisant l'ordre alphabétique sur les solutions vues comme une suite de choix d'activités, en prenant comme ordre sur les activités l'ordre d'énumération utilisé dans l'algorithme. On a alors la solution gloutonne g qui est la plus petite solution correcte pour cet ordre par construction de l'algorithme.

Soit Opt la solution optimale la plus petite pour cet ordre. On a donc $Opt \geq g$.

* Si $Opt = g$ c'est fini.

* Sinon, on appliquant le propriété d'échange comme ci-dessus et on construit une solution encore optimale et plus petite que Opt : on aboutit à une contradiction !

Un raisonnement simple

Soit le problème suivant : n tâches sont à exécuter séquentiellement. On connaît la durée prévisionnelle de chaque requête. On cherche à minimiser le temps total d'attente des utilisateurs.

On cherche donc une permutation des n tâches qui minimise ce temps d'attente. Le critère glouton naturel est d'ordonner les tâches par durée croissante. On est dans le cas particulier où une solution consiste en fait à un ordre -ou une permutation-. Supposons qu'une solution ne soit pas triée par durée croissante. On va montrer qu'elle n'est pas optimale. Si elle n'est pas triée par durée croissante, il existe deux tâches A et B consécutives dans la solution telles que la durée de A est strictement supérieure à celle de B. Soit alors la solution obtenue en permutant A et B : pour toutes tâches différentes de A et de B le temps d'attente ne change pas. Pour A le temps d'attente est augmenté de la durée de B. Pour B il est diminué de la durée de A : comme la durée de A est strictement supérieure à celle de B, le temps total d'attente a été strictement diminué donc la solution n'est pas optimale, q.e.d.

Les pièces de monnaie

On dispose des pièces de monnaie correspondant aux valeurs $\{a_1, \ldots, a_n\}$, avec $1 = a_1 < a_2 \ldots < a_n$. Pour chaque valeur le nombre de pièces est non borné. Etant donnée une quantité c entière, on veut trouver une façon de "rendre" la somme c avec un nombre de pièces minimum.

L'algorithme glouton naturel est le suivant :

Input: c somme à payer entière >0
les valeurs faciales des pièces
$1 = a_1 < a_2 ... < a_n$
Output: le nb de chaque pièces pour payer c en minimisant le nb total de pièces
$nbPieces \leftarrow 0$;
for $(int\ i = n; i \geq 1; i--)$ **do**
  $n\_i \leftarrow c \div a\_i$;
  $nb\_pieces \leftarrow nb\_pieces + n\_i$;
  $c \leftarrow c\ modulo\ a\_i$;
end

ou, plus proche du schéma glouton "classique" :

Input: c somme à payer entière >0
les valeurs faciales des pièces
$1 = a_1 < a_2 ... < a_n$
Output: le nb de chaque pièces pour payer c en minimisant le nb total de pièces
int nbtotal $\leftarrow 0$; //solution vide
int[n] nb ;
for $(int\ i = n - 1; i \geq 0; i--)$ **do**
  $nb[i] \leftarrow c/a[i]$;
  $nb\_pieces \leftarrow nb\_pieces + nb[i]$;
  $c \leftarrow c\ modulo\ a[i]$;
end

Cet algorithme glouton donnera-t-il toujours la solution optimale ? Cela dépend en fait des valeurs des a_i. Etudions deux cas :

1. Soit $n = 3, a_1 = 1, a_2 = 5, a_3 = 7$: pour $c = 10$ l'algo glouton utilisera 4 pièces (1 de 7 et 3 de 1), alors que la solution optimale n'en utilise que deux (de 5). Donc, dans ce cas, l'algorithme glouton n'est pas optimal.

2. Par contre, soit $n = 3, a_1 = 1, a_2 = 2, a_3 = 5, a_4 = 10$.

La solution produite par l'algorithme glouton est `c div 10, c mod 10 div 5, c mod 10 mod 5 div 2, c mod 10 mod 5 mod 2 div 1`, soit plus simplement `c div 10, c mod 10 div 5, c mod 5 div 2, c mod 5 mod 2`.

Soit o_{10}, o_5, o_2, o_1 une solution optimale. On a donc $c = 10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1$.

Alors, $o_1 < 2$: sinon on remplace deux pièces de 1 par une de 2.

De même, $o_2 < 3$: sinon, on remplace trois pièces de 2 par une de 5 et une de 1.

De plus si $o_2 = 2$, on a $o_1 = 0$: sinon on remplace deux pièces de 2 et une de 1 par une de 5.

Enfin $o_5 < 2$: sinon, on remplace deux pièces de 5 par une de 10.

Donc nécessairement, $5 * o_5 + 2 * o_2 + o_1 < 10$, $o_{10} = c\ div\ 10 = g_{10}$ et $5 * o_5 + 2 * o_2 + o_1 = c\ mod\ 10$.

Comme $o_1 < 2$ et $o_2 < 3$ et $(o_2 = 2 \Rightarrow o_1 = 0)$, $2 * o_2 + o_1 < 5$: donc $o_5 = c\ mod\ 10\ div\ 5 = g_5$.

On a donc $2 * o_2 + o_1 = c\ mod\ 10\ mod\ 5 = c\ mod\ 5$ et comme $o_1 < 2$, $o_2 = c\ mod\ 5\ div\ 2 = g_2$ et $o_1 = c\ mod\ 5\ mod\ 2 = g_1$.

Donc la solution gloutonne est optimale. C'est même l'unique solution optimale.