Algorithmes Gloutons (greedy algorithms)

AAC

Sophie Tison
Université Lille 1
Master Informatique 1ère année

Bilan du dernier cours: Quand peut-on utiliser la programmation dynamique?

. La solution (optimale) d'un problème de taille n s'exprime en fonction de la solution (optimale) de problèmes de taille inférieure à n -c'est le principe d'optimalité-.

. Une implémentation récursive "naïve" conduit à calculer de nombreuses fois la solution de mêmes sous-problèmes.

Comment utiliser la programmation dynamique?

. On définit une table pour mémoriser les calculs déjà effectués: à chaque élément correspondra la solution d'un et d'un seul problème intermédiaire, un élément correspondant au problème final.

. On peut avoir une version itérative ou une version récursive...

. La table contient les coûts ou valeurs des solutions: une fois la table remplie, une "remontée" dans la table permettra de construire la solution.

Quelques exemples classiques de programmation dynamique

Beaucoup d'algorithmes dans des domaines divers -recherche opérationnelle, apprentissage, bio-informatique, commande de systèmes, linguistique, théorie des jeux, processus de Markov....- utilisent le principe de la programmation dynamique.

Quelques exemples classiques de programmation dynamique

- l'algorithme de Warshall-Floyd (clôture transitive d'une relation, plus court chemin dans un graphe),
- l'algorithme de Viterbi,
- le calcul de la distance de deux chaînes (cf *diff* d'Unix) et plus généralement les problèmes d'alignement de séquences,
- l'algorithme de Cocke-Younger-Kasami (CYK),
- l'algorithme d'Earley (analyse syntaxique de phrases),
- les arbres binaires optimaux, la triangulation d'un polygone...

Trois paradigmes d'algorithmes

- Programmation Dynamique
- Algorithmes Gloutons
- Diviser pour régner

Le principe d'une méthode gloutonne

Avaler tout ce qu'on peut
=
Construire au fur et à mesure une solution en faisant les choix qui paraissent optimaux localement

Dans certains cas, cela donnera finalement la meilleure solution: on parlera d'algorithmes gloutons exacts.

Dans d'autres, non, on parlera d'heuristiques gloutonnes.

Le cadre général

...est souvent celui des problèmes d'optimisation.

On cherche à construire une solution à un problème qui optimise une fonction objectif.

Exemple: un problème de gestion de algol es.

.Une instance I du problème sera les horaires des cours à assurer et les salles;

.Une solution éventuelle sera une affectation de salle à chaque cours,

.Une solution correcte vérifie les contraintes -il n'y a pas deux cours en même temps dans une salle... voire, il y a un quart d'heure de battement...-

.Une solution optimale optimise une fonction objectif, par exemple, qui minimise le nombre maximum de salles utilisées.

UN PREMIER EXEMPLE: LE MONNAYEUR

On dispose des pièces de monnaie correspondant aux valeurs $\{a_0, \ldots, a_{n-1}\}$, avec $1 = a_0 < a_2 \ldots < a_{n-1}$.

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.

UN PREMIER EXEMPLE: LE MONNAYEUR

Une instance du problème est la donnée des valeurs faciales des pièces $\{a_0, \ldots, a_{n-1}\}$, avec $1 = a_0 < a_2 \ldots < a_{n-1}$ et de la somme c à payer.

Une solution est pour chaque type de pièce ($0 \leq i \leq n-1$) un nombre de pièces nb_i.

Elle est correcte si $\sum_{i=0}^{n-1} nb_i * a_i = c$: on a bien payé c (exactement);

Elle est optimale si $\sum_{i=0}^{n-1} nb_i$ est minimal: on a minimisé le "coût", ici le nombre total de pièces.

UNE INSTANCE DU MONNAYEUR

- Les pièces $a_0 = 1, a_1 = 2, a_2 = 5, a_3 = 10, a_4 = 20$
- la somme à payer $c = 26$

Solution optimale?

1 pièce de 20,
pas de pièce de 10,
1 pièce de 5,
pas de pièce de 2,
1 pièce de 1.

Peut-on faire mieux?

Pourquoi?

Comment justifier que la solution est optimale

Pour les pièces $a_1 = 1, a_2 = 2, a_3 = 5, a_4 = 10, a_5 = 20$ et la somme à payer $c = 26$, la solution: $20, 5, 1$ est-elle optimale? Preuve par cas?

- Si on utilise 1 pièce de 20, il reste 6 à payer: au moins deux pièces!
- Si on utilise 0 pièce de 20, il reste 26 à payer en 10, 5, 2, 1: au moins 3 pièces

Donc c'est optimal!

Quel algorithme?

```
//c somme à payer entière >0
//valeurs  des pièces: 1=a[0] < a[1] ... < a[n-1]
//les a[i] triés en croissant
{
int nb_pieces=0;//solution vide
int[n] nb; //initialisé à 0
i=n-1;
 while (c>0)  {//solution non complète
     if c>=a[i]
        {nb[i]++;nb_pieces++;c=c-a[i];}
        //ajout 1 pièce a_i
      else i--;   //on passe à la suivante
    }
 }
```

Ou:

```
//c somme à payer entière >0
//valeurs  des pièces: 1=a[0] < a[1] ... < a[n-1]
//les a[i] triés en croissant : critère glouton
{int nbtotal=0;//solution vide
int[n] nb;
 for (int i=n-1; i>=0; i--)
   {nb[i]=c div a_i;
    nb_pieces=nb_pieces+nb[i];
    c=c mod a_i};
}
```

L'algorithme est-il correct?

Par exemple, soit 26 à payer en pièces de 7, 6, 1

algorithme glouton: 3 Pièces de 7, 5 pièces de 1

une meilleure solution: 4 Pièces de 6, 2 pièces de 1

solution optimale: 2 pièces de 7, 2 pièces de 6

Donc, Non, l'algorithme n'est pas correct!

Quand est-il correct?

Par exemple, est-il toujours correct pour $1, 2, 5, 10, 20$?

La solution produite par l'algorithme glouton est

$g_{20} = c \div 20$
$g_{10} = (c \mod 20) \div 10$
$g_5 = ((c \mod 20) \mod 10) \div 5$ soit $g_5 = (c \mod 10) \div 5$
$g_2 = ((c \mod 10) \mod 5) \div 2$ soit $g_2 = (c \mod 5) \div 2$
$g1 = (c \mod 5) \mod 2$

QUE DIRE DE L'OPTIMALE?

Soit $o_{20}, o_{10}, o_5, o_2, o_1$ une solution optimale. On a
$c = 20 * o_{20} + 10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1$

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

De même, $o_2 < 3$, sinon, on remplace 3 pièces de 2 par 1 de 5 et 1 de 1. De plus si $o_2 = 2$, on a $o_1 = 0$, sinon on remplace 2 pièces de 2 et 1 de 1 par 1 de 5;

$o_5 < 2$, sinon, on remplace 2 pièces de 5 par 1 de 10.

$o_{10} < 2$, sinon, on remplace deux pièces de 10 par une de 20.

donc
$10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1 <= 10 + 5 + 4 < 20$

Que dire de l'optimale?

$c = 20 * o_{20} + 10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1$ avec

$10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1 < 20$

Donc $o_{20} = c \div 20 = g_{20}$ et
$10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1 = c \ mod \ 20$.

De même, $5 * o_5 + 2 * o_2 + o_1 < 10$ donc
$o_{10} = c \ mod \ 20 \div 10 = g_{10}$.
Donc $5 * o_5 + 2 * o_2 + o_1 = c \ mod \ 10$;

Comme $2 * o_2 + o_1 < 5$, $o_5 = c \ mod 10 \div 5 = g_5$ et
$2 * o_2 + o_1 = c \ mod \ 5$

De même, $o_2 = (c \ mod) \ 5 \div 2 = g_2, o_1 = c \ mod \ 5 \ mod \ 2 = g_1$.
Donc la solution gloutonne est (l'unique) optimale.

Le principe général d'une méthode gloutonne

Comme on l'a dit plus haut, on est 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 à notre 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 général

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.

Le schéma d'algorithme est alors:

```
//on va construire la solution dans Sol
//initialisation
Ens=E;
Sol.Init //= ensemble (ou suite) "vide" ou..
tant que Non Sol.Complete? et Ens.NonVide? {
 select(x,Ens); //choix x selon critère glouton
 si Sol.AjoutPossible(x) alors Sol.Ajout(x);fsi;
  //dans certains problèmes,toujours le cas
 si CertainesConditions alors Ens.Retirer(x);
  //selon les cas, x considéré qu'une fois
  //ou jusqu'à qu'il ne puisse plus etre ajouté
fin tant que;
//la Solution partielle est complète ..
retourne Sol
```

Quelques remarques

- 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...

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 oériode 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.

La formalisation du problème

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 : $i \in J, j \in J, i \neq j \Rightarrow d_i >= f_j \; ou \; f_i <= d_j$

. On a satisfait le maximum de demandes, i.e. $|J|$ (le cardinal de J) est maximal.

Quel critère glouton?

- Par durée croissante? i.e. de la plus courte à la plus longue?
- Par date de début croissante, i.e. de la première -qui commence le plus tôt- à la celle qui commence le plus tard?
- Par date de fin croissante, i.e. de la première -qui finit le plus tôt- à celle qui finit le plus tard?
- Par date de début décroissante?

L'ALGORITHME GLOUTON

```
{g=Vide;
 Lib=0; //la ressource est dispo
 nb=0; //pas de résa prise en compte pour le moment
   pour chaque demande i
                    dans l'ordre croissant de fin
    si d_i>=Lib
   //on peut la planifier
    {g.ajouter(i);//j'ajoute à g la demande i
        Lib=f_i;
      nb++;}
   fsi;
 fin pour;
}
```

Comment prouver que l'algorithme est correct?

1. La solution obtenue est correcte, i.e. les réservations retenues sont bien compatibles: facile à vérifier par induction.
   On prend pour invariant { *les demandes de g sont compatibles et finissent toutes au plus tard à Lib*}.

2. La solution est optimale...??

Comment prouver que l'algo est correct?

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.

$dist(A, B) = 2^{-i}$

PREUVE DE L'OPTIMALITÉ DE L'ALGO:

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:

Les trois cas

1. $g_i \neq o_i$
2. o_i n'est pas définie
3. g_i n'est pas définie

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'algo glouton l'aurait examinée avant g_i et l'aurait choisie.

Mais alors, si on transforme o en o' en remplaçant l'activité 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!

C'est ce qu'on appelle la propriété d'échange.

Cas 2 et 3

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!

Fin de la preuve

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!

Glouton versus Dynamique

Dans les deux cas on peut se représenter l'ensemble des solutions sous forme d'un arbre, les solutions étant construites incrémentalement et pouvant être vues comme une suite de choix.

Dans le cas de la programmation dynamique, on parcourt toutes les solutions mais on remarque que de nombreux noeuds de l'arbre correspondent aux mêmes sous-problèmes et l'arbre peut donc être élagué, ou plutôt représenté de façon beaucoup plus compacte comme un graphe (DAG: directed acyclic graph).

Dans le cas d'un algorithme glouton, on construit uniquement et directement-sans backtracking- une -et une seule- branche de l'arbre qui correspond à une solution optimale.

Complexité:

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 donc 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 de type "échange":

Propriété 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.

La Théorie

Les algorithmes gloutons peuvent être formalisés en utilisant la théorie des matroïdes.

Exemples classiques

- 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...

Bilan:

Pour mettre au point un algorithme glouton, il faut donc:

- 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: souvent dur!
- L'implémenter: en général facile et efficace!