Complexité (Quatrième année)

TD7

Maud MARCHAL

13 avril 2011

Concevoir un algorithme

Trois paradigmes :

- Diviser pour Résoudre
- Programmation dynamique
- **Algorithmes gloutons**

Algorithmes gloutons

13 avril 2011

Plan

Cadre général

Le cadre général est souvent celui des problèmes d'optimisation : on cherche à construire une solution à un problème qui optimise une fonction objectif.
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 ;
- Une fonction objectif est associée à chaque solution : on cherche une solution qui maximise (ou minimise) cette fonction objectif.

Principe d'une méthode gloutonne

- Avaler tout ce qu'on peut = Faire le choix qui paraît optimal localement
- Deux types de cas :
  - Cas où l'on obtient la meilleure solution : **algorithmes gloutons exacts**
  - Autres cas : **heuristiques gloutonnes**

Algorithme de la stratégie gloutonne : notations

Soit :

- un problème de taille n ;
- $X = [x_1 \cdots x_m]$ avec $m = O(n^k)$ la forme de toute solution ;

On recherche les solutions dans un ensemble S. A l'étape i, on a l'algorithme suivant :

```
procedure chercher_glouton(int i)
    y : elements de Si
debut
     y:= choix(Si)
      si PP(X,i,y) alors
        X[i] :=y;
        si candidat(i) alors
          SUCCES;
        sinon
          chercher_glouton(i+1);
      sinon  ECHEC
fin
```


Algorithme de la stratégie gloutonne

Avec les notations suivantes :

- $PP(X, i, y)$: vrai si $[X[1] \cdots X[i-1], y)$ vérifie le prédicat partiel ;
- candidat(i) : vrai si $[X[1] \cdots X[i-1], X[i])$ est solution ;
- La fonction choix(S_i) fait un choix définitif dans l'ensemble S à l'étape i (sans retour en arrière : on ne parcourt qu'une branche de l'arbre de recherche).
  - **choix** est un critère local de sélection des éléments de E pour construire une solution optimale, c'est souvent une fonction ad-hoc de coût polynomial.
  - Théorie des algorithmes gloutons (greedy algorithms) : les algorithmes sont basés sur la théorie des matroïdes ... que nous n'étudierons pas dans ce cours !

Complexité de la stratégie gloutonne

La complexité est en général de l'ordre de : $nlog(n) + nf(n) + ng(n)$

- nlogn : le tri selon le critère ;
- n*f(n) si f est le coût de la vérification de la contrainte et de l'ajout d'un élement ;
- n*g(n) si g est le coût de PP(X,i,y).

Les algorithmes gloutons sont donc le plus souvent efficaces.

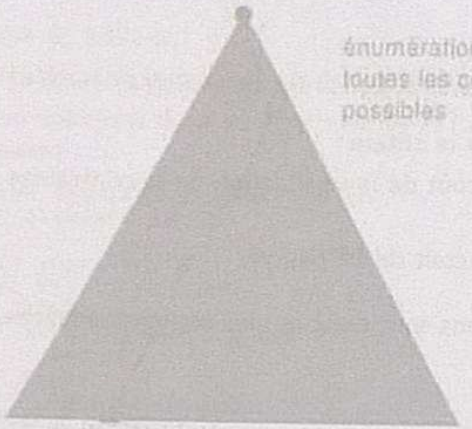
Les différents algorithmes gloutons

L'algorithme glouton peut être :

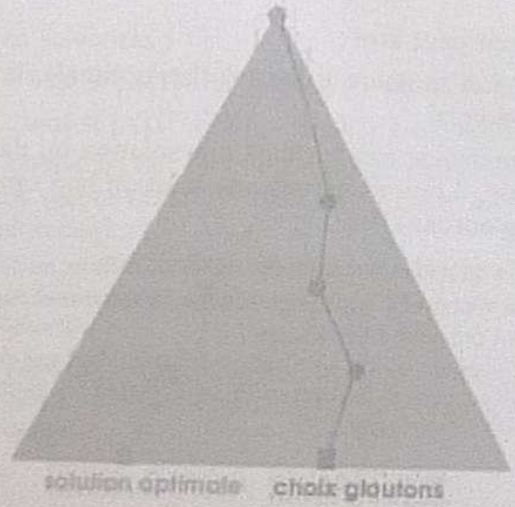
- **exact** : il conduit toujours à une solution optimale, le problème traité est alors polynomial ;
- **inexact** : il ne délivre pas toujours une solution ou dans le cas d'un problème valué, il fournit parfois une solution non optimale (heuristique gloutonne).

  Un algorithme glouton inexact est intéressant :
  - lorsque la recherche par essais successifs est beaucoup trop coûteuse pour que l'on puisse l'envisager ;
  - parce qu'il peut fournir une valeur initiale *Vopt* qui dans la recherche par essais successifs permet ensuite d'élaguer. Il peut aussi permettre de calculer une heuristique.

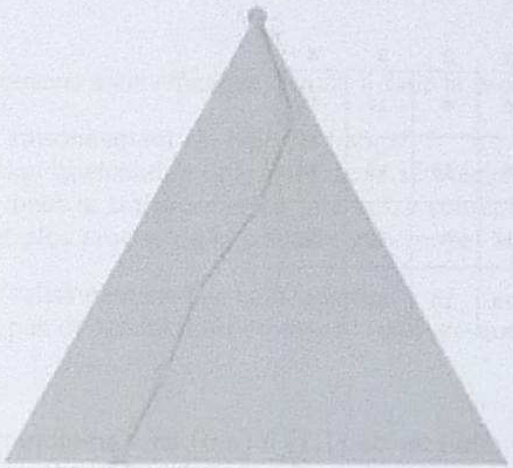
Stratégie gloutonne



Stratégie gloutonne



Stratégie gloutonne



Les exemples classiques d'algorithmes gloutons

- Algorithme de calcul d'un arbre de recouvrement d'un graphe de poids minimal : algorithmes de Prim et Kruskal ;
- Algorithme des plus courts chemins dans un graphe : algorithme de Dijkstra ;
- Code de Huffman ;
- Problèmes d'affectation et d'ordonnancement des tâches.

Premier exemple : traversée de la matrice

- Soit la matrice T :

	1	2	3	4
1	0	4	17	16
2	8	19	15	14
3	12	4	17	11
4	20	10	13	1

- On veut se déplacer de (1,1) à (n,n) en n'employant que les mouvements $\rightarrow$ et $\downarrow$. Le coût d'un chemin est la somme des valeurs contenues dans les cases empruntées.

Premier exemple : traversée de la matrice

Questions :

1. Proposez un algorithme glouton ;
2. Donne-t-il toujours une solution ?
3. Est-elle toujours optimale ?

Optimalité de la stratégie gloutonne

Les algorithmes gloutons sont efficaces... mais il faut le prouver :

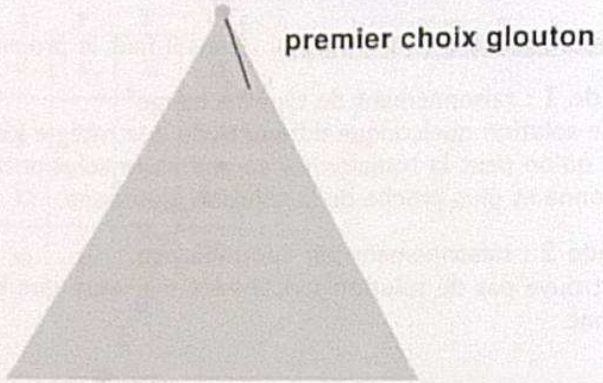
- **Méthode 1** : raisonnement de type "échange" :
  Soit une solution quelconque différente de la stratégie gloutonne : on montre qu'on peut la transformer en une autre solution au moins aussi bonne et plus proche de la solution gloutonne.
- **Méthode 2** : raisonnement par contradiction :
  On ne trouve pas de solution strictement meilleure que la solution gloutonne.

Optimalité de la stratégie gloutonne

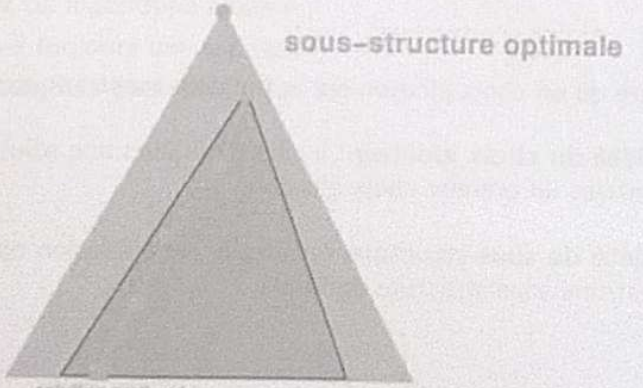
On démontre qu'un choix glouton est optimal en montrant que :

- **Propriété du choix glouton** : il existe toujours une solution optimale qui contient un premier choix glouton.
- **Propriété de sous-structure optimale** : une solution optimale contient une sous-structure optimale.

Optimalité de la stratégie gloutonne



Optimalité de la stratégie gloutonne



Optimalité de la stratégie gloutonne

Exemple avec le rendu de monnaie :

- Données :
  - s : somme à atteindre
  - M : valeurs des pièces de monnaie
- Objectif : constituer la somme s avec le moins de pièces possible
- Exemple : M={1,2,5} et s=10
  Solution optimale : 5+5 (2 pièces)

Optimalité de la stratégie gloutonne

Exemple avec le rendu de monnaie :

- Algorithme glouton :
  choisir des pièces tant que la somme n'est pas atteinte.
- Heuristique de choix : la pièce de valeur la plus grande (inférieure à la somme restant à compléter).

Optimalité de la stratégie gloutonne

Exemple avec le rendu de monnaie :
[Propriété du choix glouton]

- Si $s \geq 5$, toute solution optimale contient au moins une pièce de 5 : c'est le choix glouton.
- Si s vaut 1, 2, 3 ou 4, l'algorithme fait un bon premier choix (la valeur 2 ou la valeur 1).

Conclusion : toute solution optimale contient donc un choix glouton.

Optimalité de la stratégie gloutonne

Exemple avec le rendu de monnaie :
[Propriété de sous-structure optimale]

- Soit S une solution optimale pour s, $C \in S$ un choix glouton (la plus grosse pièce de S).
  Alors $S' = S - C$ est une solution optimale pour $s - valeur(C)$.
- Preuve : Par l'absurde :
  soit S'' meilleure que S' pour $s - valeur(C)$, alors $S'' \cup C$ est meilleure que $S' \cup C$ pour s. Contredit l'hypothèse que $S' \cup C$ est optimale.

Optimalité de la stratégie gloutonne

Exemple avec le rendu de monnaie :
[Preuve de l'optimalité]
P_n : caractérise les problèmes dont les solutions optimales comportent au plus n pièces.
Preuve par récurrence sur n :

- n=1 :
  le choix glouton produit des solutions optimales pour les problèmes P_1 (la solution se réduit à la plus grande pièce de valeur inférieure ou égale à s).
- Hypothèse : l'algorithme glouton produit des solutions optimales pour les problèmes P_n,
  Induction : l'algorithme glouton produit des solutions optimales pour les problèmes $P_{n=1}$
  Preuve : soit S une solution optimale de taille $n+1$.
  Soit $C \in S$ la plus grosse pièce de valeur inférieure à s (propriété du choix glouton), alors $S - C$ est optimale de taille n pour $s - valeur(C)$ (propriété de sous-structure optimale). Par hypothèse, l'algorithme glouton produit aussi une solution optimale S' pour $s - valeur(C)$.
  $S' \cup C$ est une solution optimale pour s correspondant au choix glouton.

Plan

Un premier exemple : gestion de la réservation

- **Problème** : n demandes de réservation (salle, équipement, etc.) sont effectuées, avec pour chacune d'entre elles l'heure du début et de la fin (on suppose qu'on n'a pas besoin d'une période de battement entre 2 réservations).
- **Objectif** : on cherche à donner satisfaction au maximum de demandes.

Formalisation du problème

Le problème est défini par :

- n : le nombre de réservations ;
- $(d_1, f_1), \cdots, (d_n, f_n)$ le début et la fin de chaque réservation ;
- En sortie : les réservations retenues, i.e. $J \subset [1..n]$ telles que :
  - elles sont compatibles : $i \in J, j \in J, i \neq j \Rightarrow d_i \geq f_j$ ou $f_i \leq d_j$;
  - on a satisfait le maximum de demandes, i.e. le cardinal de J est maximal.

Algorithme glouton

```
g=Vide;
Lib=0; // la ressource est disponible
nb=0; // pas de reservation pour le moment
pour chaque demande i dans l'ordre croissant de fin
    si d_i >= Lib
    // on peut la planifier
      g.ajouter(i); // ajout de la solutions
      Lib = f_i;
      nb++;
```


Comment prouver que l'algorithme est correct ?

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

Preuve de l'optimalité

- On définit une notion de distance sur les solutions :
  - Soient deux solutions A et 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

- On raisonne par l'absurde : supposons que la solution gloutonne ne soit pas optimale.
- Soit alors o une solution optimale telle que $dist(g, o)$ soit minimale (une telle solution o existe bien, l'ensemble des solutions étant fini).
- Soit i tel que $dist(g, o) = 2^{-i}$, on a alors 3 possibilités :
  - $g_i \neq o_i$
  - o_i n'est pas définie
  - g_i n'est pas définie

Preuve de l'optimalité

Cas $g_i \neq o_i$:

- La demande o_i a alors une date de fin au moins égale à celle de g_i sinon comme elle est compatible avec les précédentes, l'algorithme 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 contradiction !
- Remarque : on appelle cette propriété la **propriété d'échange**.

Preuve de l'optimalité

Cas o_i n'est pas définie :

- alors $|o| < |g|$: contradiction car o ne serait pas optimale.

Cas g_i n'est pas définie :

- impossible car l'activité o_i étant compatible avec les précédentes, l'algorithme glouton l'aurait choisie.

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

Arbre de recouvrement minimal : exemple

Problème de réseaux :

n villes sont reliées par des routes.
On désire poser le long de certaines routes des lignes téléphoniques de sorte que toute ville puisse joindre toutes les autres villes.

L'objectif est de minimiser la longueur totale des câbles.

Définition

Soit G=(V,E) un graphe connexe non orienté, V ensemble des n sommets, E ensemble des m arêtes.
G graphe valué : w : $E \rightarrow \mathbb{R}$ valuations sur les arêtes.

Définition : Un arbre couvrant minimum (ACM) est un sous-graphe connexe sans cycle G'=(V',E') avec $V' = V$ et $card(E') = n - 1$ qui minimise :

$$w(G') = \sum_{(u,v) \in E'} w(u,v)$$

Algorithme ACM

Calcul d'un ACM par stratégie gloutonne : la suite des minimums locaux (ajout d'une arête minimale à chaque étape) produit un minimum global.

```
procedure ACM(G)
  E' = vide
  while E' n'est pas un arbre couvrant (n-1 etapes)
      ajouter une arete minimale qui ne compromet pas E'
  end while
end procedure
```

Il existe au moins 2 algorithmes gloutons exacts pour résoudre ce problème (cf cours de graphes) :

1. Algorithme de Prim
2. Algorithme de Kruskal

Algorithme de Prim

Algorithme de Prim (1957)
Principe : l'algorithme consiste à choisir arbitrairement un sommet et à faire croître un arbre à partir de ce sommet. L'augmentation se fait de la manière la plus optimale possible.

- Etape initiale : on choisit un sommet quelconque (1 par exemple) et $G' = (1, \emptyset)$;
- A chaque étape, on enrichit le sous-arbre partiel G' en sélectionnant une nouvelle arête u=(x,y) telle que :
  - $x \in V'$, $y \in V - V'$
  - u a un poids minimum parmi les arêtes sortant de V'.
- Arrêt quand V'=V (ou |E'|=n-1).

Algorithme de Prim : application sur l'exemple

Algorithme de Prim : complexité

Algorithme de Kruskal

Algorithme de Kruskal (1956)
Principe : l'algorithme consiste à d'abord ranger par ordre de poids croissant les arêtes d'un graphe, puis à retirer une à une les arêtes selon cet ordre et à les ajouter à l'ACM cherché tant que cet ajout ne fait pas apparaître de cycle dans l'ACM.

- Etape initiale : graphe vide $G' = (\emptyset, \emptyset)$;
- A chaque étape, on ajoute l'arête de valeur minimale ne créant pas de cycle dans G'.
- Arrêt |E'|=n-1.

Algorithme de Kruskal : application sur l'exemple

Algorithme de Kruskal : complexité

Algorithme de Kruskal : optimalité

L'algorithme est exact (il donne une solution optimale).
Preuve (par contradiction) :
Soit T l'arbre donné par l'algorithme et U un arbre plus court.

- Soit a une arête dans U qui n'est pas T. L'ajout de a permet la suppression d'une arête b de T pour retrouver un arbre T'=T-{b}+{a}.
- L'algorithme est glouton : pour a, il existe b telle que $longueur(b) \leq longueur(a)$. En conséquence, l'ajout de a ne peut pas diminuer la longueur de l'abre.
- En ajoutant les arêtes de U de la même manière, on obtient U qui ne peut pas être plus court que T.

Cycle Hamiltonien : exemple du voyageur de commerce

Exemple :

Algorithme glouton : (cf. algorithme de Kruskal) :

1. Trier les arêtes selon leur valuation ;
2. Ajouter l'arête suivante de la liste, si pas de cycle.

Exemple du voyageur de commerce

Questions :

1. Y-a-t-il toujours une solution (ie un cycle hamiltonien) ?
2. L'algorithme glouton de Kruskal trouve-t-il une solution s'il en existe ?

Plan

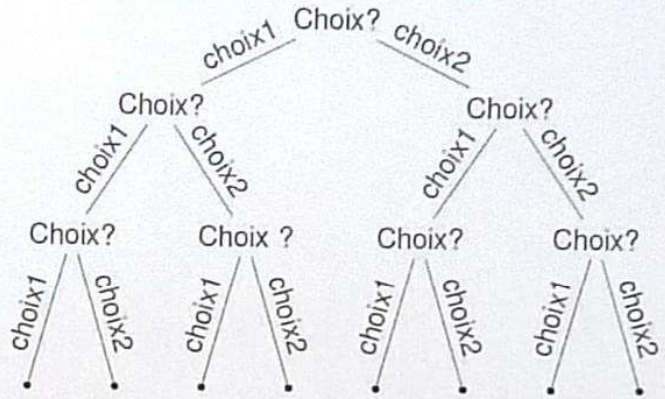
Bilan

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'est à dire que la solution est optimale : souvent dur !
- L'implémenter : en général facile et efficace !

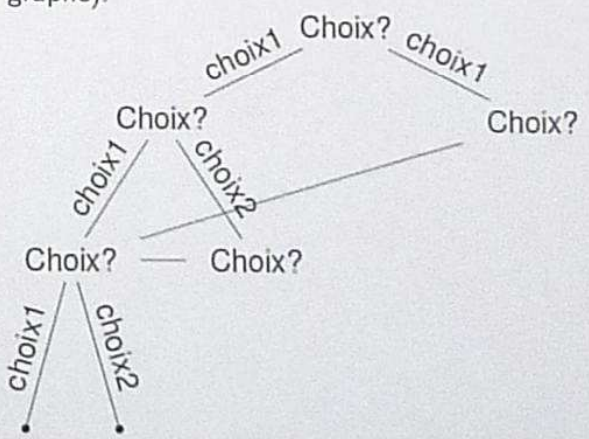
Glouton vs. Programmation dynamique

- Pour les 2 paradigmes, on peut représenter l'ensemble des solutions sous forme d'un arbre, les solutions étant construites incrémentalement et pouvant être vues comme une suite de choix.



Programmation dynamique

- 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).



Algorithme Glouton

- 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.