Chapître VI.
Stratégie gloutonne

1. Méthodes de conception des algorithmes (efficaces)

-) Diviser pour régner

-) Programmation dynamique

-) Algorithme glouton (Greedy algorithm)

2. Problème d'optimisation

La programmation dynamique est en général appliquée aux problèmes d'optimisation.

-) Dans un problème d'optimisation il y a de nombreuses solutions possibles.

-) Chaque solution possible est affectée d'une valeur.

-) On souhaite trouver une solution ayant la valeur optimale (minimum ou maximum).

Exemple:

MAXIMUM CLIQUE (Clique Maximum)

-) donnée: graphe non-orienté $G = (V, E)$

-) solution: $C \subseteq V$ est une solution si C est une clique de G, c'est-à-dire quelque soit deux sommets $x \in C$ et $y \in C$ alors x et y sont adjacents.

-) valeur: $val(C) = |C|$

3. Propriétés générales d'un algorithme glouton

Stratégie gloutonne ou algorithme glouton signifie:

> **A chaque étape, l'algorithme fait un choix optimal localement et continue de considérer uniquement ce sous-problème.**

Propriétés des algorithmes gloutons:

-) très puissants

-) simple à décrire et à implémenter

-) ne fonctionne pas toujours correctement

$\Rightarrow$ Une **démonstration de validité** est absolument nécessaire.

4. Stratégie gloutonne et des heuristiques

On appelle **heuristique** un algorithme qui est (typiquement) très puissant mais en général il ne détermine pas la valeur optimale.

Pour une heuristique (typiquement) on ne peut pas garantir une bonne qualité de valeur sortie.

On utilise la stratégie gloutonne aussi pour la conception des heuristiques parce qu'un algorithme glouton est très puissant.

Mais un algorithme glouton peut-être donne une valeur très mauvaise, c'est-à-dire très éloignée de l'optimum.

Nous considérons uniquement des algorithmes gloutons qui sortent toujours la valeur optimale d'entrée, c'est-à-dire que l'algorithme est **valide**.

Voici trois algorithmes de graphe bien-connus qu'on obtient en utilisant la stratégie gloutonne:

-) l'algorithme de DIJKSTRA qui calcule les plus courts chemins d'une source unique dans un graphe orienté

-) l'algorithme de KRUSKAL et l'algorithme de PRIM qui calculent un arbre couvrant de coût minimal dans un graphe non-orienté

Remarque. En général, ce n'est pas juste de dire qu'il n'y a uniquement qu'une stratégie gloutonne pour un problème particulier.

5. Le problème du choix d'activités

Donnée:

-) Ensemble $S = \{1, 2, \ldots, n\}$ de n **activités** concurrentes

-) Chaque activité i possède un **horaire de début** d_i et un **horaire de fin** f_i, avec $d_i \leq f_i$.

Problème:

On souhaite trouver un **choix d'activité**

compatible $S' \subseteq S$ ayant la taille optimale.

Convention: On suppose quelque soit $i, j \in S$: $d_i \neq f_j$. On peut s'imaginer que chaque activité commence à une heure pile et se termine cinq minutes avant une heure pile.

Définition. Soit $i, j \in S$ et $i \neq j$. Les activités i et j sont **compatibles** si les intervalles $[d_i, f_i]$ et $[d_j, f_j]$ ne se superposent pas.

$\Rightarrow$ Les activités i et j sont compatibles ssi

(a) $f_i < s_j$ ou (b) $f_j < s_i$.

6. L'analyse du problème

Observation:

1) On observe qu'on peut choisir les activités dans l'ordre des horaires.

2) Glouton: Si on arrive à une heure où la ressource est disponible on prend l'activité compatible avec le choix courant qui termine le plus tôt.

Remarque. L'algorithme suivant est basé sur ces deux observations bien que ça ne soit pas évident.

7. L'algorithme glouton

L'algorithme glouton résolvant le problème du choix des activités suppose que les activités entrées sont triées par ordre croissant des horaires de fin:

$$f_1 \leq f_2 \leq f_3 \leq \cdots f_n$$

Si tel n'est pas le cas, on peut les trier en $O(n \log n)$.

Le pseudo-code suppose que les entrées s et f sont représentées sous forme de tableaux.

```
CHOIX − ACTIVITE − GLOUTON(s, f)
    A ← {1}
    j ← 1
    pour i ← 2 à n
        faire si d_i > f_j
                alors A ← A ∪ {i}
                      j ← i
    retourner A
```

L'ensemble A collecte les activités selectionnées.

La variable j indique l'ajout le plus récent à A.

Comme les activités sont considérées par ordre croissant des horaires de fin, f_j est toujours l'horaire de fin maximum de toute activité présente dans A. Autrement dit,

$$f_j = \max\{f_k : k \in A\}.$$

$CHOIX - ACTIVITE - GLOUTON(s, f)$

1 $A \leftarrow \{1\}$
2 $j \leftarrow 1$
3 **pour** $i \leftarrow 2$ **à** n
4   **faire si** $d_i > f_j$
5   **alors** $A \leftarrow A \cup \{i\}$
6             $j \leftarrow i$
7 **retourner** A

Le **choix optimal localement** est fait aux lignes $4 - 5$:

Si l'activité i et à dernière activité choisie j sont compatibles alors l'algorithme ajoute l'activité i à l'ensemble du choix A.

Question: Pourquoi faut-il trier les f_i tel que

$$f_1 \leq f_2 \leq f_3 \leq \cdots f_n?$$

Remarque. Parfois on trouve une étape de tri au début d'un algorithme glouton pour créer un "bon ordre" de l'entrée.

Exemple: L'entrée est déjà triée.

$1 = [2, 3.5]$, $2 = [1, 4.5]$, $3 = [1, 5.5]$,
$4 = [3, 5.5]$, $5 = [5, 7.5]$, $6 = [7, 8.5]$,
$7 = [5, 8.5]$, $8 = [8, 10.5]$

Nous constatons que l'algorithme est vraiment très puissant.

8. Analyse du temps et démonstration de la validité

$CHOIX-ACTIVITE-GLOUTON$ est très efficace. Elle peut ordonnancer un ensemble S de n activités en $\Theta(n)$, en supposant que les activités soient déjà triées selon leur horaire de fin.

La prochaîne activité choisie par $CHOIX-ACTIVITE-GLOUTON$ est toujours celle ayant l'horaire de fin le moins tardif. L'activité choisie est donc un choix <u>glouton</u> au sens où, intuitivement, il laisse le plus de chances possibles aux activités restantes d'être ordonnancées.

Le choix glouton est celui qui maximise la quantité de temps libre restant.

Les algorithmes gloutons ne produisent pas toujours une solution optimale. Pourtant, $CHOIX-ACTIVITE-GLOUTON$ trouve toujours une solution optimale à une instance du problème du choix d'activités.

La méthode de démonstration de la validité d'un algorithme glouton est classique:

1) On suppose qu'il y a une entrée pour laquelle l'algorithme glouton trouve une solution s_a qui n'est pas optimale.

2) Soit s_o une solution optimale de cette entrée.

3) On fait un certain échange entre les deux solutions possible s_a et s_o pour obtenir une nouvelle solution s qui n'existe pas à cause de notre choix de solution optimale s_o, c'est une contradiction.

Théorème. *L'algorithme* $CHOIX-ACTIVITE-GLOUTON$*, qui s'exécute en temps* $\Theta(n)$*, donne les solutions de la plus grande taille possible pour le problème du choix d'activités.*

Démonstration: Supposons qu'il existe une entrée $S = \{1, 2, \ldots, n\}$ à ordonnancer où la sortie $A = \{i_1, i_2, \ldots, i_r\}$ de l'algorithme n'est pas optimale.

Soit $A' = \{i'_1, i'_2, \ldots, i'_s\}$ une solution optimale d'entrée S, c'est-à-dire $r < s$.

Supposons que les activités des deux solutions sont triées tel que

$$f_{i_1} \leq f_{i_2} \leq f_{i_3} \cdots f_{i_r} \text{ et}$$

$$f_{i'_1} \leq f_{i'_2} \leq \cdots f_{i'_s}.$$

Evidemment $A \neq A'$. Soit $q(A')$ le plus petit indice q pour lequel $i_q \neq i'_q$. Maintenant nous reconsidérons le choix de la solution optimale.

Important: Soit A' une solution optimale d'entrée S pour laquelle l'indice $q(A')$ est le plus grand possible.

Plan: Construisons une nouvelle solution optimale A'' tel que $q(A') < q(A'')$ et on aboutira à une contradiction.

Cas 1: $q(A') = r + 1$

$\Rightarrow A \subseteq A'$ et $A' = \{i_1, i_2, \ldots, i_r, i'_{r+1}, \ldots, i'_s\}$

$\Rightarrow$ Quand l'algorithme considère l'activité $i = i'_{r+1}$, la dernière activité selectionnée est $j = i_r$.

Parce que $f_{i_r} < s_{i'_{r+1}}$ l'algorithme sélectionne i'_{r+1} bien que $i'_{r+1} \notin A$: c'est une contradiction.

Cas 2: $q(A') \le r$

Soit $A'' = (A' \setminus \{i'_{q(A')}\}) \cup \{i_{q(A')}\}$.

C'est-à-dire que nous remplaçons l'activité $i'_{q(A')}$ de solution optimale par le choix glouton $i_{q(A')}$ de l'algorithme.

Affirmation:

$A'' = \{i_1, i_2, \ldots, i_{q(A')}, i'_{q(A'+1)} \cdots, i'_s\}$ *est une solution optimale. Alors* $q(A'') = q(A') + 1$ *et* $q(A'') > q(A')$ *contredisent le choix de* A'.

Ce qui démontre le théorème. $\square$

L'algorithme choisit $i_{q(A')}$ bien que $i'_{q(A')} \in A'$ et $i_{q(A')-1} = i'_{q(A')-1} \in A'$ sont compatibles.

$$\Rightarrow \qquad f_{i_{q(A')}} \leq f_{i'_{q(A')}}.$$

Nous constatons que $i'_{q(A')} \in A'$ et $i'_{q(A')+1} \in A'$ sont compatibles.

$$\Rightarrow \qquad f_{i'_{q(A')}} < s_{i'_{q(A')+1}}$$

$$\Rightarrow \qquad f_{i_{q(A')}} < s_{i'_{q(A')+1}}$$

C'est-à-dire que $i_{q(A')}$ et $i'_{q(A')+1}$ sont compatibles.

Alors $A'' = \{i_1, i_2, \ldots, i_{q(A')}, i'_{q(A'+1)} \cdots, i'_s\}$ est une solution possible. En plus A'' est une solution optimale parce que $|A''| = |A'|$.

Finalement $q(A'') = q(A') + 1$ à cause de notre construction: c'est une contradiction.

9. Graphes d'intervalles et ensemble stable maximum

Notre problème est aussi connu sous le nom **ensemble stable maximum d'un graphe d'intervalles**.

On peut créer un graphe d'intervalles dont les sommets sont les activités et dont les arêtes relient les activités incompatibles. C'est-à-dire que deux sommets i et j sont adjacents ssi $[d_i, f_i] \cap [d_j, f_j] \neq \emptyset$.

Trouver le plus grand nombre de sommets dans un ensemble stable, revient à trouver le plus grand nombre d'activités compatibles entre elles.

Remarque. L'algorithme $CHOIX-ACTIVITE-GLOUTON$ est un algorithme efficace pour le problème NP-complet **Maximum Independent Set (Ensemble stable maximum)** pour la classe des graphes d'intervalles.

10. Algorithme glouton contre programmation dynamique

Il existe **deux caractéristiques** qu'un problème peut avoir, et qui se prètent à une stratégie gloutonne:

1) Propriété du choix glouton

On peut arriver à une solution globalement optimale en effectuant un choix localement optimal (glouton).

2) Sous-structure optimale

Une solution optimale du problème contient la solution optimale du sous-problème.

En programmation dynamique, on fait un choix à chaque étape, mais ce **choix dépend de la solution des sous-problèmes**.

Dans un algorithme glouton, on fait **le choix qui semble le meilleur sur le moment**, puis on résout les sous-problèmes qui surviennent une fois que le choix est fait.

La programmation dynamique et la stratégie gloutonne exploitent la propriété de sous-structure optimale.

En général un algorithme glouton est plus rapide. Bien entendu, **il faut démontrer qu'un choix glouton à chaque étape engendre une solution optimale globalement** et c'est là qu'un peu d'astuce peut s'avérer utile.

11. Problème du sac-à-dos

Maximum Knapsack (Sac-à-dos Maximum)
(version discrète)

-) donnée: $S = \{1, 2, \ldots, n\}$ ensemble de n objets

Chaque objet i vaut v_i francs et pèse w_i kilogrammes.

On ne peut pas porter plus de W kilos dans le havresac.

-) solution: $S' \subseteq S$ est une solution si

$$\sum_{i \in S'} w_i \leq W$$

-) valeur: $val(S') = \sum_{i \in S'} v_i$

On ne peut pas prendre une partie d'objet. Puisqu'on peut prendre "rien" (zero) ou "tous" (un), on appèle cette version **discrète**.

Pour la version discrète on peut obtenir un algorithme efficace par programmation dynamique.

Variante fractionnaire du problème du Sac-à-dos

On suppose qu'on peut prendre des fractions d'objets.

Observation: $\frac{v_i}{w_i}$ est la valeur par kilo d'objet i

Algorithme glouton: "On prend toujours l'objet qui a la plus grande valeur par kilo jusqu'à ce que le sac-à-dos est rempli."

1) On trie les objets tel que

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$$

2) On prend les objets dans l'ordre $1, 2, \ldots$ jusqu'à ce que le sac-à-dos est rempli.

3) Si nécessaire on prend une fraction du dernier objet.

Question 1: Pourquoi est-ce que cet algorithme glouton ne trouve pas toujours une solution optimale pour la version discrète?

Question 2: De quelle façon s'exécute l'algorithme de programmation dynamique pour la version discrète?

12. D'autres méthodes de conception des algorithmes

-) Backtracking

-) Branch & Bound

On utilise ces deux méthodes de conception des algorithmes en particulier pour la conception des algorithmes pour des problèmes NP-complet ("difficile").

En général on ne peut pas obtenir des algorithmes efficaces utilisant le Backtracking ou le Branch & Bound.