Algorithmes Gloutons (greedy algorithms)

Le cadre: On est souvent dans le cadre des problèmes d'optimisation: on cherche à construire une solution à un problème qui optimise une fonction objectif:

Pour une instance I (par exemple, les horaires des cours à assurer, des salles), trouver une solution (par exemple une affectation de salle à chaque cours) qui vérifie les contraintes (par exemple, il n'y a pas deux cours en même temps dans une salle...) et qui optimise une fonction objectif (par exemple, minimise le nombre de salles).

Idée intuitive: Le principe est très simple!

Dans le cas d'une méthode gloutonne, on construit tout simplement cette solution incrémentalement en rajoutant à chaque pas un élément selon un critère glouton, i.e. celui qui nous paraît "localement" le meilleur- un choix à "court terme". Si l'algorithme nous donne toujours la bonne solution, on parlera d'algorithme glouton exact sinon d'heuristique gloutonne.

La construction de la solution est donc souvent très simple, le problème étant plutôt de justifier que la solution construite ainsi est bien optimale!

On peut comparer avec le principe de la programmation 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 comprimé.

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.

Le schéma général

Le problème: 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 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:

```
//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);        //on choisit x selon critère glouton
 si Sol.AjoutPossible(x) alors Sol.Ajout(x); fsi;
  //dans certains problèmes, c'est toujours le cas
 si CertainesConditions alors Ens.Retirer(x);
  //selon les cas, x ne sera  considéré qu'une fois
  // ou  jusqu'à qu'il ne puisse plus etre ajouté
fin tant que;
//la Solution partielle est complète ...normalement
retourne 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 le schéma général: 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!

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 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'algorithme 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 .

On peut donc ainsi montrer par contradiction qu'il ne peut y avoir de solution strictement meilleure que la solution gloutonne: soit une solution optimale la plus proche possible de la solution gloutonne.-comme l'ensemble des solutions est fini, il y en a une- Si ce n'est pas la solution gloutonne, on peut transformer cette solution en une aussi bonne -donc toujours optimale- et plus proche de la gloutonne!

Souvent pour mesurer la "proximité" entre deux solutions, on trie les solutions 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. Plus formellement, on peut souvent définir un ordre total sur les solutions $<$ telle que *Gloutonne* soit maximale (ou minimale) parmi les solutions. On montre ensuite que si *Gloutonne* $> S$ alors, par la propriété d'échange, il existe S', *Gloutonne* $>= S' > S$, avec S' aussi bonne que S. Soit S maximale pour l'ordre $<$ parmi les solutions optimales (elle existe, l'ensemble des solutions étant fini): d'après ce qui précède, Gloutonne est S et donc est optimale.

Remarque: ce schéma de preuve a l'avantage d'etre 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.

Théorie: Les algorithmes gloutons sont basés sur la théorie des matroïdes...que nous n'étudierons pas.

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!