Optimisation par séparation et évaluation (*branch and bound)*

Dans ce TP nous présentons la méthode algorithmique de *séparation et évaluation* qui est l'analogue du *backtracking* pour les problèmes d'optimisation.

On suppose qu'on souhaite construire un objet x qui maximise une fonction de coût $f(x)$. On suppose aussi que l'objet x peut se construire par étapes, ce qu'on notera $x = (x_1, x_2, \dots, x_n)$.

Une méthode *branch and bound* cherche à construire x itérativement : si on suppose qu'on a déjà débuté la construction de $x = (x_1, x_2, \dots, x_k)$ on envisage toutes les continuations possibles de construction et on garde celle de valeur maximale. Ainsi, on peut déterminer x par un parcours de profondeur de l'arbre de recherche en gardant à chaque nœud le fils donnant le meilleur coût. Ceci constitue la partie *séparation* (*branch*) de la méthode.

La partie *évaluation* (*bound*) consiste à *estimer* à chaque nœud le meilleur cout que l'on pourrait obtenir. Si ce coût est inférieur à une solution déjà trouvée précédemment lors du parcours, alors il ne sert à rien d'étudier cette partie de l'arbre : on l'élague.

Ainsi cette technique permet avec certitude de déterminer une solution de coût maximal mais sans avoir à explorer exhaustivement toutes les solutions du problèmes.

1 Problème du sac à dos

On considère n objets $\{x_0, \dots, x_{n-1}\}$ ayant chacun un poids $p(x_i)$ et une valeur $v(x_i)$ qui sont des entiers. On rappelle que le problème du sac à dos consiste pour une capacité de sac C fixée à déterminer l'ensemble d'objets à choisir, de valeur maximale, mais dont le poids total ne dépasse pas C.

En OCaml, un tel problème sera codé par le type

```
type sacados {
    poids : int list;
    valeur : int list;
    capacite : int
};;
```

On pourra tester ses solutions avec l'instance problème :

```
let s1 = {
    poids = [200; 314; 198; 500; 300];
    valeur = [40; 50; 100; 95; 30];
    capacite = 1000
};;
```

Une solution au problème du sac à dos est une liste de valeurs 0 ou 1, où 0 signifie qu'on prend l'objet et 1 qu'on ne prend pas l'objet.

Question 1

Écrire une fonction `verifie : sacados -> int list -> bool` vérifiant si une solution est correcte (mais pas nécessairement optimale).

Question 2

Écrire une fonction `cout : sacados -> int list -> int` retournant le cout d'une solution donnée.

Le problème du sac à dos consiste donc à déterminer une solution correcte de coût maximum.

Question 3

Écrire une fonction `range : sacados -> int -> sacados` qui choisit le k-ième objet et le place dans le sac à dos : on obtient alors un nouveau problème de sac à dos avec un objet de moins et dans lequel la capacité du sac a diminué.

1.1 Une majoration par relaxation

Pour pouvoir appliquer la méthode *branch and bound* pour une maximisation, il faut pouvoir à tout instant majorer la valeur optimale que l'on peut obtenir. On procède en général par **relaxation des contraintes** : on va considérer le même problème mais avec des contraintes plus souples : ici, on va se permettre de prendre des fractions d'objets. On peut imaginer par exemple que les objets de base étaient des gateaux et qu'on peut se permettre de prendre seulement une part d'objet. On appelle ce problème le *sac à dos fractionnaire*.

Il est facile de remarquer que dans ce cas, un algorithme glouton permet de déterminer le sac à dos optimal :

- on prend à chaque fois l'objet de meilleur rapport valeur/poids et on le place dans le sac à dos;
- si l'objet en question ne rentre pas, on ne prend qu'une fraction de ce dernier objet, ce qui termine de remplir le sac.

Question 4

Écrire une fonction `majorer : sacados -> int` retournant la valeur optimale du sac à dos dans le cas du problème du sac à dos fractionnaire.

Evidemment, cette valeur optimale est un majorant de tous les coûts qu'on peut obtenir pour un sac à dos fractionnaire. Or, un sac à dos dans le problème non relaxé est un cas particulier de sac à dos fractionnaire, donc la majoration obtenue est toujours valide même pour le problème initial.

1.2 Séparation et évaluation

Question 5

Implémenter la stratégie *branch and bound* sous forme d'une fonction récursive :

`branchbound : sacados -> int -> (int * int list)`

telle que `branchbound s vmax` retourne

- un couple `(v, sol)` où `sol` est une solution optimale au problème du sac à dos et `v` la valeur correspondante
- `(-1, [])` s'il a été estimé que ce sac à dos ne pourra pas donner une meilleure valeur que `vmax`. Dans ce cas, aucun appel récursif à `branchbound` ne sera effectué (élagage).

Question 6

En déduire une fonction `resoudre : sacados -> (int * int list)` résolvant le problème du sac à dos avec la méthode par séparation e t évaluation.

2 Voyageur de commerce

Dans le problème du voyageur de commerce on considère un ensemble de n villes $x_0, \dots, x_{n-1}$ séparées entre elles par une distance entière. On appelle *tournée* une liste d'entiers entre 0 et $n-1$ de longueur n telle que chaque entier apparaît exactement une fois dans la liste; cette liste représente un ordre de visite de chacune des villes. Le coût d'une tournée $t = [a_0; \dots; a_{n-1}]$ est la distance totale parcourue pour effectuer la tournée et revenir au point de départ :

$$\text{coût}(t) = \sum_{i=0}^{n} \text{dist}\left(x_{a_i}, x_{a_{i+1}}\right) \text{ avec pour convention } a_n = a_0$$

On peut remarquer que ce problème se modélise en termes de graphe : l'ensemble des villes forme un graphe complet non orienté dans lequel on souhaite déterminer un cycle hamiltonien de coût minimum.

Ainsi on peut représenter une instance du problème par la matrice de poids de ce graphe :

```
type tsp = int array array;
```

par exemple :

```
let g1 = [|
[| 0; 3; 4; 2; 7|];
[| 3; 0; 4; 6; 3|];
[| 4; 4; 0; 5; 8|];
[| 2; 6; 5; 0; 6|];
```

```
[| 7; 3; 8; 6; 0|]|];
```

Question 7

Écrire une fonction `poids : tsp -> int list -> int` retournant le poids d'une tournée.

2.1 Une minoration simple

Le problème du voyageur de commerce étant cette fois ci un problème de minimisation, on cherche à obtenir à chaque nœud de l'arbre de recherche une *minoration* de la solution qu'on peut obtenir : cela permet d'élaguer ce nœud, dans le cas où la minoration est plus grande qu'une solution déjà déterminée.

Pour le voyageur de commerce on peut utiliser cette méthode simple : le coût d'une tournée est égal à 1/2 de la somme pour chaque sommet des poids des 2 arêtes utilisées par ce sommet (en effet on a compté chaque sommet deux fois). Ainsi, une tournée aura toujours un poids d'au moins 1/2 de la somme des poids des deux plus légères arêtes adjacentes à chaque sommet.

D'autres méthodes de minoration plus sophistiquées existent, mais on se contentera de cette minoration rudimentaire.

Question 8

Écrire une fonction `minorer : tsp -> int list -> int` telle que `minorer g c` retourne un minorant de la tournée qu'on peut obtenir sachant qu'on a débuté par le chemin `c`.

2.2 Séparation et évaluation

Question 9

En déduire un programme en OCaml déterminant une tournée optimale pour le problème du voyageur de commerce avec la méthode de séparation et évaluation.