TP8-Algorithmes probabilistes et d'approximation

15 janvier

1 Algorithmes probabilistes

On donne ici quelques éléments de syntaxe C et Ocaml au sujet de la génération (pseudo-)aléatoire de nombres dans ces deux langages. Dans les deux cas, il faut commencer par initialiser le générateur de nombres pseudo-aléatoires (PRNG : pseudorandom number generator) à l'aide d'une graine. Il y a deux façons de faire cette initialisation, à choisir selon le comportement souhaité :

- Initialiser le générateur avec une graine connue fixe. Dans ce cas, à chaque exécution de votre code, les choix aléatoires effectués seront toujours les mêmes et votre algorithme probabiliste se comportera comme un algorithme déterministe. Ce comportement est utile lors des phases de debug.
- Initialiser le générateur avec une graine qui changera à chaque exécution. Dans ces conditions, votre algorithme probabiliste fera des choix différents à chaque exécution (c'est le comportement souhaité une fois le debug effectué). En Ocaml, l'appel à `Random.self_init ()` permet d'obtenir ce comportement. En C, on fournit généralement au générateur pseudo aléatoire la date courante pour l'obtenir ; cette dernière se récupère à l'aide de `time(NULL)` et nécessite d'importer `time.h`.

Récapitulatif des commandes qui peuvent vous être utiles :

Action	Ocaml	C
Initialiser le générateur avec une graine fixe s	`Random.init s`	`srand(s)`
Initialiser le générateur avec une graine changeante	`Random.self_init ()`	`srand(time(NULL))`
Tirer un entier entre 0 inclus et n exclus	`Random.int n`	`rand() % n`
Tirer un booléen	`Random.bool ()`	`rand() % 2 == 0`
Tirer un flottant entre 0 et 1 inclus	`Random.float 1.`	`(float)rand() / RAND_MAX`

Pour retrouver les commandes adaptées en C, il suffit de se souvenir du fonctionnement de la fonction `rand` : cette fonction ne prend pas d'entrée et génère un entier aléatoirement entre 0 et une valeur maximale fixée dont le nom est `RAND_MAX` et dont la valeur dépend de la machine.

1. On va commencer par implémenter le tri rapide probabiliste de type Las Vegas. Pour cela, on a besoin d'une fonction de partition d'un tableau entre deux cases données autour d'un pivot donné qui renvoie l'indice du tableau où se trouvera le pivot à la fin de l'opération de partition. On rappelle le principe de la méthode de Hoare :

   On crée deux indices de parcours initialisés respectivement au début et à la fin de la zone à partitionner. Ces indices vont se rejoindre en échangeant au fur et à mesure les éléments de gauche de valeurs supérieures au pivot et les éléments de droite de valeurs inférieures au pivot. Ici on crée deux indices i et j et on fait respecter l'invariant suivant : "les termes d'indices `deb` à $i-1$ sont inférieurs à la valeur de pivot et ceux entre $j+1$ et `fin` lui sont supérieurs". On ne progressera que tant que $i \leq j$.

   Ecrire une fonction `partition : int array->int->int->int` telle que

`partition tab i j pivot` effectue la partition autour de la valeur `tab[pivot]` entre les indices `i` et `j` du tableau et renvoie l'indice du tableau où se retrouve la valeur `tab[pivot]`. On supposera garanti le fait que $i \leq pivot \leq j$.

2. En déduire une fonction `randomquick_sort : int array->unit` qui trie un tableau passé en entrée en utilisant le tri rapide randomisé.

3. Ecrire une fonction `test_product : in array array -> int array array->int array array->bool` qui teste si le produit de deux matrices carrées A et B est bien la troisième matrice passée en argument. On implémentera la stratégie probabiliste vue en classe.

2 BinPacking

Le problème dit BinPacking est défini ainsi :

Instance : un entier naturel C et une famille $X = x_0, \ldots, x_{n-1}$ d'entiers naturels

Solution admissible : une partition de X en $B_0 \sqcup \cdots \sqcup B_{k-1}$ telle que $\sum_{x \in B_i} x \leq C$ pour tout i

Optimisation : minimiser k

Autrement dit, on nous donne n *objets* de *volumes* $x_0, \ldots, x_{n-1}$, et l'on dispose de *boîtes* de *capacité* C. Il faut répartir les objets dans k boîtes de manière à ce que la somme des volumes reste toujours inférieure à la capacité, en minimisant le nombre k de boîtes utilisées.

Remarque 1 *Des problèmes de ce type se posent en particulier quand on s'intéresse à l'allocation mémoire (quand on veut implémenter* `malloc`*, essentiellement) : les boîtes représentent les zones mémoire (contiguës) disponibles, les objets les demandes d'allocation. Une difficulté supplémentaire dans ce cas est qu'on doit décider comment traiter chaque objet dès qu'il arrive, sans savoir quels autres objets arriveront plus tard (on parle d'algorithme* online*).*

1 Donner une solution optimale pour BinPacking sur l'instance suivante : $C = 10$ et $X = 2, 5, 4, 7, 1, 3, 8$.

2.1 Caractère NP-complet (pour le td)

On admet (*cf* exercice TD réductions) que le problème SubsetSum, défini comme suit, est NP-complet :

Instance : un multi-ensemble A d'entiers naturels et un entier $s > 0$

Question : existe-t-il $B \subseteq A$ tel que $\sum_{x \in B} x = s$.

2 Définir le problème de décision BPD associé au problème d'optimisation BinPacking.

3 On considère le problème Partition :

Instance : n entiers naturels $x_0, \ldots, x_{n-1}$

Question : Existe-t-il $I \subseteq [0 \ldots n-1]$ tel que $\sum_{i \in I} x_i = \sum_{i \notin I} x_i$?

Montrer que Partition est NP-complet. *On pourra partir d'une instance de* SubsetSum *et lui ajouter* $2s$ *et* $\sum_{i=0}^{n-1} x_i$.

4 En déduire que BPD est NP-complet.

2.2 Stratégies gloutonnes

On va considérer dans ce sujet trois stratégies gloutonnes pour le problème BinPacking. On suppose à partir de maintenant que les poids des différents objets sont majorés par C, puisque sinon il n'y a clairement aucune solution.

- La stratégie **next-fit** considère les objets dans l'ordre d'arrivée, et ajoute ces objets dans la boîte courante tant que c'est possible. Quand ce n'est plus le cas, on ferme (définitivement) cette boîte et l'on en crée une nouvelle, qui devient la boîte courante.
- La stratégie **first-fit** considère aussi les objets dans l'ordre d'arrivée, mais maintient une liste (initialement vide) de boîtes $B_0, \ldots, B_{k-1}$. À chaque fois que l'on considère un objet, on cherche le premier i tel que l'objet rentre dans la boîte B_i :
  - s'il en existe un, on ajoute l'objet dans cette boîte ;
  - sinon, on crée une nouvelle boîte B_k dans laquelle on place l'objet.
- La stratégie **first-fit-decreasing** procède comme **first-fit** mais commence par trier les objets par ordre décroissant de volume.

On représente une instance de BinPacking par un entier `capacity` (la capacité C des boîtes) et une liste d'entiers correspondant aux poids des objets ; on supposera que tous les poids sont inférieurs ou égaux à la capacité.

```
type instance = int * int list
```

5 Proposer un type `box` pour représenter une boîte (et son contenu). On souhaite pouvoir déterminer le volume disponible dans la boîte, et y ajouter un objet, en temps constant.

6 Écrire une fonction `next_fit` qui résout le problème en utilisant la stratégie *next-fit*.

```
val next_fit : instance -> box list
```

7 Écrire de même une fonction `first_fit`.

8 Écrire une fonction `first_fit_decreasing`.

9 Quelle solution obtient-on pour l'instance de la question 1 avec les différentes stratégies ?

10 Déterminer la complexité dans le pire cas des fonctions qui correspondent aux différentes stratégies.

11 **À ne traiter qu'à la fin.** On considère la variante suivante de *first-fit-decreasing*, appelée *best-fit-decreasing* : on commence par trier les objets par volume décroissant, puis à chaque étape on ajoute l'objet à la boîte *la plus remplie* dans laquelle il rentre (ou l'une quelconque des telles boîtes en cas d'égalité). On crée une nouvelle boîte si nécessaire.
Proposer un algorithme (qu'on ne demande pas d'implémenter) permettant d'appliquer cette stratégie avec une complexité $(n \log n)$.

2.3 Analyse des approximations (à faire en TD ou quand tout le code est terminé)

2.4 Analyse de *next-fit*

On note m le nombre de boîtes utilisées par la stratégie *next-fit* sur une instance $C, X = (x_0, \ldots, x_{n-1})$ et m^* le nombre optimal de boîtes sur cette même instance. On numérote B_0, B_{m-1} les boîtes utilisées par *next-fit*, dans l'ordre de leur utilisation et l'on note v_i le volume total des objets présents dans la boîte v_i et $V = \sum_{i=0}^{n-1} x_i$.

12 Montrer que $v_i + v_{i+1} > C$ pour $0 \leq i < m - 1$.

13 En déduire que *next-fit* fournit une 2-approximation pour BinPacking.

2.5 Analyse de *first-fit-decreasing*

On reprend les notations de la partie précédente, avec m le nombre de boîtes utilisées par *first-fit-decreasing* et m^* le nombre optimal de boîtes. On note $x_0 \geq \cdots \geq x_{n-1}$ les poids.

On considère la boîte B_j avec $j = \lfloor \frac{2m}{3} \rfloor$, et l'on note x le volume maximal d'un objet rangé dans la boîte B_j.

Cas $x > C/2$

14 Montrer que $m \leq \frac{3}{2} m^*$.

Cas $x \leq C/2$ On note v le plus grand volume disponible (en fin d'exécution) dans les boîtes $B_0, \ldots, B_{j-1}$.

15 Montrer que $\sum_{l=j}^{m-1} v_l > v + 2v(m - j - 1)$.

16 Conclure.

2.6 Difficulté de l'approximation (en TD)

17 Soit $\epsilon > 0$. Par réduction de Partition, montrer que s'il existe un algorithme fournissant en temps polynomial une $(\frac{3}{2} - \epsilon)$-approximation pour BinPacking, alors $P = NP$.

3 N-reines : backtracking et Las Vegas

On s'intéresse au problème des N reines. Il s'agit de placer sur un échiquier de taille $N \times N$, N reines sans qu'elles puissent se prendre l'une l'autre. Cette partie du TP sera à traiter en C. Une solution du problème sera représentée par un tableau de taille N où la case d'indice k contiendra le numéro de la colonne où se trouve la reine de la ligne k s'il en contient une (en effet une ligne contient au plus une reine).

On rappelle que la reine se déplace en ligne, en colonne ou en diagonale.

On va utiliser tout d'abord un algorithme de backtracking que l'on transformera en un algorithme de type Las Vegas. Le principe est le suivant : on essaie de positionner les reines ligne par ligne en vérifiant qu'une nouvelle reine positionnée ne peut pas prendre une reine préalablement positionnée.

18 Ecrire une fonction `bool check(int n, int sol[], int k)` qui teste si la position de la reine dans la ligne `k` est compatible avec celles des reines en lignes `0` à `k-1`. On suppose que les `k` premières lignes sont convenablement remplies et on vérifie si la `k`eme prolonge la solution partielle ou non. La solution en cours de construction est stockée dans le tableau `sol`.

19 Ecrire une fonction `bool solve(int n, int sol[], int k)` qui teste si une solution partielle avec une grille remplie jusqu'à la ligne `k-1`, stockée dans `sol`, est prolongeable. La fonction sera récursive et complètera la solution au passage.

20 On veut tirer une colonne possible de manière uniforme dans l'ensemble des colonnes acceptables pour la ligne `k`.

On utilise la procédure suivante :

```
t=0;
col_choisie=0;
pour chaque colonne v :
sol[k]=v;
si (check(n,sol,k)) :
     t++;
     col_choisie = v avec proba 1/t
si (t==0) false
sinon sol[k]=col_choisie;
```

Justifier que cette procédure va choisir une colonne compatible de manière uniforme.

21 Ecrire une fonction `bool solve_prob(int n, int sol[], int k)` qui est une adaptation de `solve` où au lieu de prendre toutes les colonnes de manière systématique, on en choisit une uniformément au hasard.

22 Est ce que ce programme va toujours renvoyer une solution quand elle existe? Ecrire une fonction de type Las Vegas qui cherche une solution au problème des N reines à partir de la fonction de la question précédente.