TP7-Algorithmes probabilistes et d'approximation

15 décembre

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.

2 Echauffement

Ecrire une fonction Caml `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.

3 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 $X = 2, 5, 4, 7, 1, 3, 8$.

3.1 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.

3.2 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.

4 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.

11 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`.

12 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.

13 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.

14 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.

15 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.

5 Retour sur les grammaires

Les fonctions à implémenter dans cette partie le seront en utilisant la syntaxe C. On utilisera les bibliothèques string.h, stdlib.h et stdbool.h.

L'algorithme de Cocke-Younger-Kasami détermine si un mot est engendré par une grammaire : on utilise la programmation dynamique afin de déterminer comment générer les portions $m[i,...j]$ du mot m. Si la grammaire admet des règles ayant une forme quelconque alors la multiplicité de ces types de règles ne permet pas d'écrire un algorithme c'est pourquoi cet algorithme repose sur le fait suivant (dont nous avons parlé en TD) :

Théorème 1 *Toute grammaire qui ne génère pas le mot vide est faiblement équivalente à une grammaire dont toutes la règles sont de la forme $X \to AB$ ou $X \to a$ où A et B sont des symboles non terminaux distincts du symbole initial et a est un symbole terminal. On appelle la grammaire dont les règles sont de cette forme, forme normale de Chomsky.*

Dans cette partie, on considère un langage algébrique L ne contenant pas ε décrit par une grammaire algébrique $G = (\Sigma, V, S, \mathcal{R})$ qu'on suppose mise sous forme normale de Chomsky.

Soit un mot $m = m_1...m_n \in \Sigma^+$. On cherche à savoir s'il appartient à $L = L(G)$. Pour ce faire, on introduit pour tous $i, j \in [\![1, n]\!]$ tel que $i \leq j$ l'ensemble $E_{i,j}$ des non terminaux de G qui engendrent le mot $m_i...m_j$. Notre objectif est donc de calculer $E_{1,n}$: l'axiome S en fait partie si et seulement si $m \in L(G)$.

1. Si $i \in [\![1, n]\!]$, expliquer comment déterminer l'ensemble $E_{i,i}$ à partir de G.
2. Si $i, j \in [\![1, n]\!]$ et $i < j$, montrer que :

$$E_{i,j} = \bigcup_{k=i}^{j-1} \{X \in V \mid X \to YZ, Y \in E_{i,k} \text{ et } Z \in E_{k+1,j}\}$$

On considère alors l'algorithme dynamique suivant :

Entrée : Une grammaire G sous forme normale de Chomsky tel que $\varepsilon \notin L(G)$ et un mot $m \neq \varepsilon$.
Sortie : Vrai si $m \in L(G)$ et faux sinon.

CYK(G, m) =
- Initialiser une matrice $E = (e_{i,j})_{i,j \in [\![1,n]\!]}$ remplie de $\emptyset$
- Pour i allant de 1 à n
  - Pour toute règle de la forme $X \to a$ de G
    - Si $a = m_i$ alors $e_{i,i} \leftarrow e_{i,i} \cup \{X\}$
- Pour d allant de 1 à $n-1$
  - Pour $e_{i,j}$ sur la d-ème surdiagonale de E
    - Pour k allant de i à $j-1$
      - Pour toute règle de la forme $X \to YZ$ de G
        - Si $Y \in e_{i,k}$ et $Z \in e_{k+1,j}$ alors $e_{i,j} \leftarrow e_{i,j} \cup \{X\}$
- Si $S \in e_{1,n}$ renvoyer vrai, sinon renvoyer faux

3. On considère la grammaire G_{ex} dont les règles sont données par :

$$S \to XY, \qquad T \to ZT \,|\, a, \qquad X \to TY, \qquad Y \to YT \,|\, b, \qquad Z \to TZ \,|\, b$$

   En utilisant l'algorithme précédent, déterminer si *abab* appartient à $L(G_{ex})$. On dessinera explicitement la matrice E et le contenu de ses cases en fin d'algorithme.

4. Déterminer sa complexité en fonction de $|m|$ et d'une autre grandeur pertinente.

La fin de cette partie fait implémenter l'algorithme de Cocke-Younger-Kasami, dont le pseudo-code est donné ci-dessus. Les terminaux seront un sous ensemble des caractères ASCII. Les non terminaux (aussi appelés variables) seront représentés par des entiers numérotés consécutivement à partir de 0. L'axiome portera systématiquement le numéro 0. Une règle dans une grammaire sous forme normale de Chomsky sera représentée à l'aide de la structure suivante :

```
struct regle{
    int type;
    int membre_gauche;
    char lettre;
    int variable1;
    int variable2;
};
typedef struct regle regle;
```

Le type d'une règle est un entier valant 1 si la règle est de la forme $X \to a$ et 2 si elle est de la forme $X \to YZ$. Dans les deux cas, le champ membre_gauche contient le numéro du non terminal X. Dans le premier cas, le champ lettre contient a et les champs variable1 et variable2, -1. Dans le deuxième cas, les champs variable1 et variable2 contiennent les numéros de Y et Z respectivement et le champ lettre un caractère ne faisant pas partie de l'ensemble des terminaux de la grammaire considérée.

Une grammaire sous forme normale de Chomsky sera représentée à l'aide de la structure :

```
struct grammaire{
    int nb_variables;
    int nb_regles;
    regle* productions;
};
typedef struct grammaire grammaire;
```

Si g est un objet de type grammaire représentant $G = (\Sigma, V, S, \mathcal{R})$, g.nb_variables correspond à $|V|$, g.nb_regles à $|\mathcal{R}|$ et g.productions est un tableau de taille g.nb_regles contenant les règles de G.

5. Indiquer les valeurs des champs nb_variables et nb_regles d'un objet de type grammaire représentant la grammaire G_{ex}. Après avoir précisé une numérotation des non terminaux, déclarer deux objets de type regle représentant les règles $S \to XY$ et $Y \to b$ de G_{ex}. Terminer en implémentant totalement la grammaire de l'exemple.

6. Ecrire une fonction `void libere_g(grammaire* g)` qui permet de libérer l'espace mémoire utilisé par g.

Pour représenter un ensemble de non terminaux d'une grammaire donnée sur un ensemble de variables V, on utilisera un tableau de taille $|V|$ rempli de booléens. A la case i, ce tableau contiendra true si la variable numéro i est présente dans l'ensemble et false sinon.

7. Ecrire une fonction bool CYK(grammaire* g, char m[]) implémentant l'algorithme de Cocke-Younger-Kasami. On pensera à libérer l'expace mémoire alloué pour faire la programmation dynamique.

8. Vérifier votre réponse à la question 4 à l'aide de cet algorithme puis déterminer si $m_1 = bbaabaabbbab$ et $m_2 = abaabaabbbab$ font partie de $L(G_{ex})$.

9. Expliquer comment modifier l'algorithme CYK de façon à ce qu'il réponde correctement au problème du mot lorsqu'on lève les contraintes $\varepsilon \notin L(G)$ et $m \neq \varepsilon$ (on suppose toujours que G est sous forme normale de Chomsky).