Programme de colle 3

Thibault Mabillotte

2 Questions de Cours exigibles

1. Connaitre l'algorithme du mélange de Knuth et la preuve de sa correction

On peut se demander comment mélanger un tableau de manière uniforme. L'algorithme du mélange de Knuth répond à ce problème.

On donne le code en C du mélange de Knuth où swap est la fonction qui échange deux cases d'un tableau et rand est la fonction qui renvoie un entier naturel de manière aléatoire.

```c
void knuth(int* tab, int n) {
  for (int i=0, i<n, i+=1) {
    int k = rand()%(i+1); // entier entre 0 et i
    swap(tab,i,k)
  }
}
```

Propriété : L'algorithme du mélange de Knuth génère une permutation du tableau de manière équiprobable.

Démonstration : Prouvons qu'après l'exécution de l'algorithme, le tableau est mélangé selon une des permutations de $[\![0, n-1]\!]$ avec probabilité uniforme, c'est-à-dire $\frac{1}{n!}$.

On pose l'invariant de boucle suivant : "après le tour numéro i de la boucle, les cases de 0 à i du tableau ont une probabilité uniforme d'être mélangées selon une des partitions σ de $[\![0, i]\!]$".

Si $i = 0$, la seule permutation possible est l'identité qui se produit donc de manière équiprobable.

Supposons que l'invariant soit vrai au i^e tour de la boucle et montrons qu'il reste vrai au tour suivant.

L'invariant étant vrai au tour i, on note σ la permutation selon laquelle sont mélangées les cases entre 0 et i après le tour i.

Au tour numéro $i + 1$, l'algorithme tire $k \in [\![0, i+1]\!]$ avec une probabilité uniforme et échange les cases k et $i + 1$.

Soit σ' une permutation de l'ensemble $[\![0, i+1]\!]$.

Montrons que la probabilité que les cases de 0 à $i + 1$ du tableau soient mélangées selon σ' est $\frac{1}{(i+2)!}$. On raisonne par disjonction de cas selon la valeur de $\sigma'(i + 1)$:

Si $\sigma'(i + 1) = i + 1$ on a la situation suivante :

0 $\quad\quad\quad\quad$ i	$i+1$
σ	$i+1$

Alors la probabilité d'obtenir la permutation σ' est :

$$P(\sigma') = P(k = i+1) \times P\left(\sigma = \sigma'_{|[\![0,i]\!]}\right)$$

k étant tiré uniformément et σ étant équiprobable parmi toutes les permutation de $[\![0, i]\!]$, on trouve :

$$P(\sigma') = \frac{1}{i+2} \times \frac{1}{(1+i)!} = \frac{1}{(i+2)!}$$

Si $\sigma'(i+1) = j < i+1$.
σ' étant une permutation, il existe $t \neq i+1$ tel que $\sigma(t) = i+1$. On a alors la situation suivante :

$$\begin{array}{c|c|c|c|c|c|c}
 & & t & & i+1 & \\
\hline
 & & i+1 & & j & \\
\hline
\end{array}$$

Dès lors, la probabilité que les cases de 0 à $i+1$ soient mélangées selon σ' est :

$$P(\sigma') = P(k = t) \times P(\sigma = \sigma'')$$

où $\sigma'' = \begin{cases} i \neq t & \mapsto & \sigma'(i) \\ t & \mapsto & j \end{cases}$ une permutation de $[\![0, n]\!]$. k étant tiré uniformément et σ étant équiprobable parmi toutes les permutation de $[\![0, i]\!]$, on trouve :

$$p(\sigma') = \frac{1}{i+2} \times \frac{1}{(i+1)!} = \frac{1}{(i+2)!}$$

L'invariant de boucle est donc valide. Au tour $i = n-1$, les n cases du tableau ont une probabilité uniforme d'être mélanger selon une des partitions σ de $[\![0, n-1]\!]$.

2. Montrer qu'on peut transformer un algorithme de Monte Carlo en algorithme de Las Vegas si on dispose d'un vérificateur et donner l'espérance de son temps d'exécution

Proposition : Si on dispose d'un algorithme de Monte-Carlo $\mathcal{A}$ avec une probabilité d'erreur p et une complexité f tel qu'il existe un vérificateur v avec complexité g qui décide si $\mathcal{A}(x)$ est bien une solution avec une complexité g alors il existe un algorithme de Las Vegas $\mathcal{A}'$ qui résout le problème avec une espérance de temps de calcul $\frac{1}{1-p}(f+g)$.

Démonstration : Montrons l'existence d'un tel algorithme $\mathcal{A}'$ et montrons qu'il a l'espérance de temps de calcul recherchée.

On considère l'algorithme $\mathcal{A}'$ suivant :

while true **do**
    $y \leftarrow \mathcal{A}(x)$
    if $v(x,y)$ **then**
        return y
    end if
end while

Notons X la variable aléatoire comptant le nombre de tours de la boucle non bornée. La boucle s'arrête dès que y est une solution valide du problème, or l'algorithme $\mathcal{A}$ à une probabilité d'erreur p donc X suit une loi géométrique de raison $1-p$.

À chaque tour de boucle, l'algorithme $\mathcal{A}'$ calcul $\mathcal{A}(x)$ ce qui a un coût f puis il vérifie que y est bien une solution du problème à l'aide de l'algorithme v ce qui a un coût g.

En moyenne, l'algorithme va faire $E(X)$ tour de boucle donc l'espérance de temps de calcul de l'algorithme est :

$$\frac{1}{1-p}(f+g)$$

3. Réduction de l'erreur dans un algorithme de Monte Carlo sans faux négatif

Définition : On appelle faux négatif un résultat faux sur une instance positive et un faux positif un résultat vrai sur une instance négative.

Sur l'exemple précédant, il n'y a pas de faux négatif par contre il y a des faux positifs.

Proposition : Si on a un algorithme de Monte-Carlo $\mathcal{A}$ sans faux négatif avec une erreur probabilité d'erreur p alors en le répétant k fois, on obtient un algorithme de Monte-Carlo avec une probabilité erreur p^k.

Démonstration : Montrons l'existence d'un algorithme de Monte Carlo sans faux négatif avec une probabilité d'erreur plus petite.

On définit l'algorithme $\mathcal{A}'$ suivant :

```
Require: x ∈ E une instance du problème
  for k ∈ ⟦1, k⟧ do
    if non A(x) then
      return Faux
    end if
  end for
  return Vrai
```

$\mathcal{A}'$ est bien un algorithme de Monte Carlo. En effet, son temps d'exécution ne dépend pas de l'aléa puisque la boucle effectue toujours k tours. En revanche, la sortie dépend de l'aléa puisque $\mathcal{A}$ est un algorithme de Monte Carlo.

Montrons que $\mathcal{A}'$ est sans faux négatif. Soit $x \in E$ une instance du problème.
Si $\mathcal{A}'(x) =$ Faux alors une exécution de l'algorithme $\mathcal{A}$ sur x a renvoyer Faux. Or $\mathcal{A}$ est sans faux négatif donc x est une instance négative du problème.
Ainsi, si $\mathcal{A}'(x) =$ Faux alors x est une instance négative du problème. Autrement dit, $\mathcal{A}'$ est sans faux négatif.

Notons V l'événement "L'algorithme $\mathcal{A}'$ renvoie vrai" et $\forall i \in [\![1, k]\!]$, V_i l'événement "la i^e exécution de l'algorithme $\mathcal{A}$ renvoie une réponse positive". Les exécution de l'algorithme $\mathcal{A}$ étant indépendantes, les V_i sont indépendants.
Si l'instance est positive alors l'algorithme renvoie Vrai avec une probabilité de 1. Il n'y a pas d'erreur possible puisque $\mathcal{A}'$ est sans faux négatif.
Si l'instance est négative alors la probabilité que $\mathcal{A}'$ renvoie Vrai est :

$$P(V) = P(V_1 \cap \cdots \cap V_k) = \prod_{k=1}^{n} P(V_k) = p^k$$

Ainsi, la probabilité d'erreur de $\mathcal{A}'$ est p^k qui est bien plus petite que celle de $\mathcal{A}$.

4. Donner un algorithm qui permet de sélectionner k éléments dans un tableau de taille n de manière uniforme

On considère un tableau de taille n dans lequel on veut tirer k éléments du tableau de manière uniforme. Pour un nombre quelconque d'éléments à récupérés, on peut réutiliser le mélange de Knuth et récupérer les k premières cases du tableau.

Montrons que cette solution permet bien de sélectionner k éléments uniformément.
Soit X est une partie de $[\![0, n-1]\!]$ de taille k. Une permutation de $[\![0, n-1]\!]$ telle que ses k premiers éléments sont les éléments de X est entièrement déterminée par :

- L'ordre des k premiers éléments de X : $!k$ choix possibles.
- L'ordre des $n-k$ éléments qui n'appartiennent pas à X : $(n-k)!$ choix possibles.

Par principe multiplicatif, il y a $k!(n-k)!$ permutations de ce type. Comme le mélange de Knuth permet d'obtenir une permutation du tableau parmi les $n!$ permutations possibles de manière équiprobable on trouve que :

$$P(X) = \frac{k!(n-k)!}{n!} = \frac{1}{\binom{n}{k}}$$

On donne une implémentation en C d'une telle sélection :

```c
int* selection(int* tab, int n, int k) {
  int* res = malloc(sizeof(int)*k);
  for (int i=0, i<k, i++) {
    res[i] = tab[k];
  }
  for (int i=0, i<n, i++) {
    int x = rand()%(i+1) ;
    if (x<k) {
      res[x] = tab[i];
    }
  }
  return res;
}
```

5.

6. Montrer qu'un algorithme glouton pour le problème d'optimisation somme partielle fournit une 1/2 approximation

Définition : On définit le problème de maximisation, dit problème de somme partielle, de la façon suivant :

entrée : $t_1, \cdots, t_n$ des entiers naturels et $C \in \mathbb{N}$.

sortie : $I \subset [\![1, n]\!]$ qui maximise $S = \sum_{i \in I} t_i$ en vérifiant que $S \leq C$.

On propose l'algorithme glouton suivant :

$S \leftarrow 0$
for all t_i trié par valeur décroissante **do**
  if $t_i + S < C$ **then**
    $S \leftarrow S + t_i$
  end if
end for
return S

Propriété : Cet algorithme renvoie une $\frac{1}{2}$ approximation du problème de somme partielle.

Démonstration : Notons S_r la somme renvoyée par l'algorithme. On veut montrer que :

$$S_r \geq \frac{1}{2} S_{\text{opt}}$$

Pour cela, on va montrer que $S_r \geq \dfrac{1}{2} C$ et par définition du problème, $C \geq S_{\text{opt}}$, ce qui permettra de conclure.

Supposons que $\sum_{i=1}^{n} t_i \leq C$. Dans ce cas l'algorithme renvoie clairement $\sum_{i=1}^{n} t_i$ qui est la solution optimale car les t_i sont tous positifs.

Supposons $\sum_{i=1}^{n} t_i > C$. On considère alors t_j, le premier élément que l'algorithme n'ajoute pas à la somme partielle. On note S cette somme partielle contenant tous les t_i de 1 à $j-1$. Par construction de l'algorithme on a :

$$S + t_j = \sum_{i=1}^{j} > C$$

Comme on considère les t_i par ordre décroissant, $t_j \leq t_{j-1}$ donc $t_j \leq S$. On peut alors faire la majoration suivant :

$$S + t_j \leq S + S = 2S$$

Or on sait que $S + t_j > C$ d'où :

$$2S > C \iff S > \frac{C}{2}$$

Par construction de la somme partielle, la solution renvoyée par l'algorithme est plus grande donc :

$$S_r \geq S > \frac{C}{2}$$

Ce qui suffit pour conclure d'après ce qu'y est écrit plus haut.

7.

8.

9.