DS n°05 - 4h

Ce sujet est composé de deux parties complètement indépendantes.

Partie I : Complexité de Kolmogoroff

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractère différentes désigne la même entité, mais du point de vue mathématique pour la police en italique (par exemple n, $\mathcal{D}$ ou π) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`, `d` ou `pi`).

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Il faudra coder des fonctions à l'aide du langage de programmation OCaml exclusivement, en reprenant l'en-tête de fonctions fourni par le sujet, sans s'obliger à recopier la déclaration des types. Il est permis d'utiliser la totalité du langage OCaml mais il est recommandé de s'en tenir aux fonctions les plus courantes afin de rester compréhensible. Des rappels ponctuels de documentation du langage OCaml peuvent être proposés à titre d'aide. Quand l'énoncé demande de coder une fonction, sauf demande explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1 Complexité de Kolmogoroff

Nous notons Σ l'ensemble ordonné des 256 caractères ASCII étendus usuels et Σ^* l'ensemble des chaînes de caractères. Pour toute chaîne de caractères $x \in \Sigma^*$, la longueur de x, notée $|x|$, est le nombre de caractères qui la composent. Par exemple, la longueur de la chaîne `"abac"` est 4. Le nombre d'occurrences d'un symbole $\sigma \in \Sigma$ dans une chaîne de caractères $x \in \Sigma^*$ est noté $|x|_\sigma$. Par exemple, $|\texttt{abac}|_\texttt{a} = 2$.

Dans l'ensemble du sujet, nous nous appuyons sur une *machine universelle*, c'est-à-dire une fonction OCaml `eval`, de type `string -> string`, telle que :

— si la chaîne x, de type `string`, est le code source d'une expression OCaml y de type `string` et si l'exécution du code x se termine sans erreur, alors `eval x` se termine et a pour valeur de retour la valeur de y ;

— sinon, `eval x` ne se termine pas.

Les exécutions de `eval` ont lieu sur une machine idéale dont la mémoire est infinie et qui est capable de gérer des types natifs de taille quelconque.

Définition : Pour toute chaîne de caractères $y \in \Sigma^*$, nous disons que la chaîne de caractères $x \in \Sigma^*$ est une *description* de la chaîne y si le calcul `eval x` se termine et renvoie la chaîne y. Nous appelons *complexité de Kolmogoroff* de y et notons $K(y)$ la longueur de la plus courte chaîne de caractères $x \in \Sigma^*$ qui décrit y.

L'objet de ce sujet est d'étudier des propriétés et diverses majorations de la complexité de Kolmogoroff. Dans toutes nos illustrations, nous nous concentrons sur la description de la chaîne de caractères

$$y_0 = \texttt{"1000000..."} \in \Sigma^*,$$

qui correspond à l'entier $10^{(10^{10})}$ écrit en base 10 et que nous fixons une fois pour toutes.

1.1 Un premier exemple

☐ 1 – Proposer une première majoration de la complexité de Kolmogoroff $K(y_0)$, qui s'appuie sur la chaîne de caractères $x_0 = y_0 =$ `"1000000..."` comme description de y_0.

Indication OCaml : Il est rappelé que la fonction `string_of_int` convertit un entier en une chaîne de caractères.

☐ 2 – Soient n un entier naturel et n' la partie entière de $\dfrac{n}{2}$. Exprimer la quantité 10^n en fonction de $10^{n'}$. Compléter le code OCaml suivant en utilisant une stratégie « diviser pour régner » :

```
let exp10 n = (* Calcul de 10^n a ecrire *)
in string_of_int (exp10 (exp10 10))
```

afin d'en faire une description de la chaîne de caractères $y_0 =$ `"1000000..."`. En déduire une nouvelle borne grossière (à 10^2 près) de la complexité de Kolmogoroff $K(y_0)$, significativement meilleure que celle de la question 1.

☐ 3 – Décrire quelle ou quelles difficultés adviendraient si l'on exécutait le code de la question 2 sur une machine réelle.

1.2 Quelques propriétés

Indication OCaml : L'expression `String.make (n : int) (sigma : char) : string` désigne la chaîne de caractères répétant n fois le caractère $\sigma \in \Sigma$. Pour tout entier n compris entre 0 et 255, `Char.chr (n : int) : char` désigne le n^{e} caractère dans la numérotation ASCII.

☐ 4 – Présenter une bijection $\varphi : \mathbb{N} \to \Sigma^*$ entre l'ensemble des entiers naturels et l'ensemble de chaînes de caractères. En écrire le code sous la forme d'une fonction OCaml `phi (n : int) : string`.

Nous prétendons avoir écrit une fonction OCaml `kolmogoroff (y : string) : int` qui calcule la complexité de Kolmogoroff $K(y)$.

☐ 5 – Écrire une fonction OCaml `psi (m : int) : int` dont la valeur de retour est l'entier

$$\psi(m) = \min\{n \in \mathbb{N}\,;\ K(\varphi(n)) \geq m\}$$

où $\varphi : \mathbb{N} \to \Sigma^*$ est la bijection définie à la question 4 et qui utilise la fonction `kolmogoroff`.

☐ 6 – Établir d'une part que, pour tout entier naturel m, on a

$$K(\varphi(\psi(m))) \geq m$$

et d'autre part que l'on a

$$K(\varphi(\psi(m))) = O(\log m).$$

Discuter l'existence de la fonction OCaml `kolmogoroff`.

Définition : Nous appelons *décompresseur* toute fonction OCaml `d : string -> string`. Pour toute chaîne de caractères $y \in \Sigma^*$ et pour tout décompresseur $\mathcal{D}$ (noté informatiquement `d`), nous disons que la chaîne de caractères z est une *description* de la chaîne y par rapport à $\mathcal{D}$ si le calcul `eval (d z)` se termine sans erreur et a pour valeur de retour y. Nous appelons *complexité de Kolmogoroff par rapport à* $\mathcal{D}$ de la chaîne y, et notons $K_{\mathcal{D}}(y)$, la longueur de la plus courte chaîne de caractères $z \in \Sigma^*$ qui décrit y par rapport à $\mathcal{D}$.

☐ 7 – Dire comment se nomme en informatique un programme qui transforme un code source dans un certain langage de programmation en un code équivalent dans un second langage.

☐ 8 – Montrer que pour tout décompresseur $\mathcal{D}$, il existe une constante entière $c_{\mathcal{D}}$ telle que, pour toute chaîne de caractères y, nous avons :

$$K(y) \leq K_{\mathcal{D}}(y) + c_{\mathcal{D}}.$$

2 Estimation de la complexité grâce au décompresseur de Huffman

Nous fixons dans cette section la chaîne de caractères $x_0 \in \Sigma^*$ suivante

```
"let rec t e=if e=0 then 1 else let n=e-1 in 10*t n in let n=t 10 in t n".
```

La chaîne x_0 contient 71 caractères, l'espace étant un caractère et les guillemets ne faisant pas partie de la chaîne.

☐ 9 – Inférer le type OCaml de l'expression dénotée par la chaîne de caractères x_0.

☐ 10 – Déterminer la valeur de l'évaluation de `x0` en tant que code source OCaml.

☐ 11 – Signaler une ou plusieurs caractéristiques du code source x_0 qui rend la lecture de ce code impénétrable par un humain.

Indication OCaml : Nous rappelons le détail de quelques fonctions du module OCaml `Hashtbl` permettant de manipuler des dictionnaires (ou tableaux associatifs) mutables.

- `Hashtbl.create (n : int) : ('a, 'b) Hashtbl.t` crée un dictionnaire vide de taille initiale n.
- `Hashtbl.add (d : ('a, 'b) Hashtbl.t) (k : 'a) (v : 'b) : unit` ajoute une association entre la clé k et la valeur v au dictionnaire d.
- `Hashtbl.find_opt (d : ('a, 'b) Hashtbl.t) (k : 'a) : 'b option` vaut `Some v` si le dictionnaire `d` possède une association entre la clé k et la valeur v et vaut `None` sinon.

☐ 12 – Écrire une fonction OCaml `count (x : string) : (char, int) Hashtbl.t` dont la valeur de retour est un dictionnaire qui associe chaque caractère $\sigma \in \Sigma$ présent dans la chaîne x à son nombre d'occurrences $|x|_\sigma$.

Nous exécutons `count x0` et obtenons le dictionnaire suivant :

`'n'`	`'t'`	`'i'`	`'l'`	`'1'`	`'c'`	`'f'`	`'h'`	`'r'`	`'s'`	`'-'`	`'*'`	`'0'`	`'='`	`'e'`	`' '`
8	8	4	4	4	1	1	1	1	1	1	1	3	4	10	19

Nous appelons z_0 la chaîne de bits correspondant à la compression de la chaîne x_0 par l'algorithme de Huffman.

☐ 13 – Dessiner un arbre de Huffman associé à la chaîne de caractères x_0. Il est recommandé de placer les feuilles de gauche à droite comme dans le tableau ci-dessus.

Nous notons $\mathcal{H}$ le décompresseur qui utilise l'arbre de Huffman de la question 13 pour transformer une chaîne de bits $z \in \{0,1\}^*$ en un mot $x \in \Sigma^*$ et renvoie la chaîne de caractères `"string_of_int (`x`)"`.

☐ 14 – Calculer la longueur $|z_0|$ de la chaîne obtenue après compression de x_0. En déduire que la complexité de Kolmogoroff de $y_0 = 10000\ldots$ par rapport au décompresseur $\mathcal{H}$ vérifie

$$K_{\mathcal{H}}(y_0) \leq 239.$$

Partie II : Couplage optimal

On s'intéresse dans cette partie à la recherche d'un *couplage optimal* dans un graphe biparti pondéré, ainsi qu'un exemple d'applications.

Définitions et notations

- On considère dans ce sujet des *graphes bipartis*, qui seront *non orientés* et *pondérés*. Un tel graphe sera noté $G = (V = X \sqcup Y, E, c)$ où $c : E \to \mathbb{R}$ associe à chaque arête son *coût*.
- On définit le *coût d'un couplage* M de G comme la somme des coûts des arêtes qui le composent :
$$c(M) = \sum_{e \in M} c(e)$$
- Un couplage M est dit *optimal* s'il vérifie les deux conditions suivantes :
  — il est de cardinalité maximale ($|M| \geqslant |M'|$ pour tout couplage M' de G) ;
  — il est de coût minimal parmi les couplages de cardinalité maximale (si $|M'| = |M|$, alors $c(M) \leqslant c(M')$).
- Dans tout le sujet, la notation xy (ou yx) désignera l'arête non orientée $\{x, y\}$. Si le graphe est orienté et que l'on souhaite parler de l'arc de x vers y, on notera systématiquement (x, y).
- On note $V_f(M)$ (ou simplement V_f si M est clair d'après le contexte) l'ensemble des sommets libres pour M (ensemble des sommets qui ne sont incidents à aucune arête de M), $X_f(M)$ l'ensemble des sommets libres de X, $Y_f(M)$ l'ensemble des sommets libres de Y.
- Un *chemin alternant* de longueur $l \geqslant 1$ pour M est une suite $v_0, \dots, v_l$ de sommets de G tels que
  — pour $0 \leqslant i < l$, $v_i v_{i+1} \in E$;
  — les v_i sont distincts – on impose qu'un chemin alternant soit élémentaire ;
  — les arêtes alternent entre M et $E \setminus M$. Autrement dit, on a soit toutes les $v_{2i}v_{2i+1}$ dans M et toutes les $v_{2i+1}v_{2i+2}$ dans $E \setminus M$, soit l'inverse.

  Un tel chemin P pourra aussi être vu comme un ensemble d'arêtes : $P = \{v_0v_1, v_1v_2, \dots, v_{l-1}v_l\}$.
- Un chemin augmentant est un chemin alternant dont les deux sommets situés aux extrémités sont libres.
- On note $A \oplus B = (A \cup B) \setminus (A \cap B)$ la *différence symétrique* de deux ensembles A et B. On rappelle quelques propriétés de cette opération, valables pour tous ensembles A, B, C :
  — $A \oplus (B \oplus C) = (A \oplus B) \oplus C$ (on notera donc $A \oplus B \oplus C$) ;
  — $A \oplus B = B \oplus A$;
  — $A \oplus B \oplus B = A$.

I Préliminaires

❑ 15 Soient M un couplage et $P = x_0, \dots, x_l$ un chemin alternant pour M. Dans chacun des cas suivants, justifier très rapidement que $M \oplus P$ est un couplage et exprimer $|M \oplus P|$ en fonction de $|M|$:

a. P augmentant (*i.e.* $v_0 \in V_f$ et $v_l \in V_f$) ;

b. $v_0 \in V_f$ et $x_{l-1}x_l \in M$;

c. $x_0x_1 \in M$ et $x_{l-1}x_l \in M$.

❑ 16 Soit M et M' deux couplages sur G tels que $|M'| = |M| + 1$. Montrer qu'il existe un chemin P augmentant pour M tel que $P \subseteq M' \oplus M$.
On énoncera clairement les différentes étapes de la démonstration, sans nécessairement donner tous les détails.

I.1 Plus courts chemins

Étant donné un graphe orienté et pondéré $G = (V, E, \rho)$, on appelle *matrice d'adjacence pondérée* de G la matrice A de dimensions $(|V|, |V|)$ telle que :

- $a_{i,j} = \rho(i,j)$ si $(i,j) \in E$;
- $a_{i,j} = +\infty$ sinon.

Informatiquement, on représentera les coefficients d'une telle matrice par des `double` en C, et l'on utilisera la constante `INFINITY` pour représenter $+\infty$. Cette constante se comporte comme on peut l'attendre :

- `INFINITY + x` vaut `INFINITY` si `x` est fini ou égal à `INFINITY` ;
- `INFINITY > x` pour tout `x` fini.

❑ 17 Écrire une fonction `make_double_matrix` prenant en entrée un entier n et un `double` x, et renvoyant une matrice de dimensions (n, n) dont tous les coefficients valent x, sous forme d'un tableau de tableaux.

```
double **make_double_matrix(int n, double x);
```

❑ 18 Écrire une fonction `free_double_matrix` prenant en entrée un entier n et une matrice de dimensions (n, n) telle que renvoyée par la fonction `make_double_matrix` et libérant toute la mémoire associée à cette matrice.

```
void free_double_matrix(double **A, int n);
```

On supposera dans la suite que l'on dispose de fonctions similaires pour des matrices à coefficients entiers :

```
int **make_int_matrix(int n, int x);
void free_int_matrix(int **A, int n);
```

❑ 19 Écrire une fonction `floyd_warshall` prenant en entrée la matrice d'adjacence pondérée A d'un graphe G, supposé sans cycle de poids négatif, ainsi que sa taille n, et renvoyant une matrice `from` d'entiers, de même dimensions que A, telle que, pour $i, j \in [0 \dots n-1]$:

- `from[i][i]` a une valeur quelconque ;
- si $i \neq j$ et s'il n'existe aucun chemin du sommet i vers le sommet j dans G, alors `from[i][j]` vaut -1 ;
- sinon, `from[i][j]` vaut k, où k est l'indice d'un sommet tel qu'il existe au moins un chemin de poids minimum de i vers j qui se termine par l'arc (k, j).

On utilisera l'algorithme de Floyd-Warshall pour effectuer ce calcul, et l'on pourra modifier librement la matrice A fournie.

```
int **floyd_warshall(double **A, int n);
```

❑ 20 Quelle est la complexité de la fonction `floyd_warshall` ?

II Algorithme des plus courts chemins successifs

II.1 Étude théorique

Si P est un chemin augmentant pour M, on définit son *coût* $c_M(P)$ par :

$$c_M(P) = \sum_{e \in P \cap \overline{M}} c(e) - \sum_{e \in P \cap M} c(e)$$

On considère alors l'algorithme suivant :

Algorithme 1 Plus courts chemins successifs

Entrées : Un graphe biparti pondéré $G = (X \sqcup Y, E)$.
Sorties : Un couplage optimal de G.

```
M ← ∅
tant que M admet un chemin augmentant faire
    Soit P un chemin augmentant de coût minimal pour M
    M ← M ⊕ P
renvoyer M
```

❑ 21 Montrer que $c(M \oplus P) = c(M) + c_M(P)$.

❑ 22 Soient M et M' deux couplages tels que $|M'| = |M| + 1$. Montrer qu'il existe un couplage N tel que $|N| = |M|$ et un chemin P augmentant pour M tels que $M' = N \oplus P$.

❑ 23 On note M_k le couplage obtenu après k étapes de l'algorithme. On considère la propriété suivante : « M_k est une couplage de cardinal k et est de coût minimal parmi tous les couplages de cardinal k ». Montrer que cette propriété est un invariant pour la boucle et en déduire que l'algorithme 1 renvoie un couplage optimal.

On considère à présent le graphe $G_M = (V \cup \{s,t\}, E_M, \lambda)$ dont les arcs et les poids sont exactement :

- les (s,x) avec $x \in X_f(M)$, avec un poids $\lambda(s,x) = 0$;
- les (y,t) avec $y \in Y_f(M)$, avec un poids $\lambda(y,t) = 0$;
- les (x,y) avec $x \in X$, $y \in Y$ et $xy \in E \setminus M$, avec un poids $\lambda(x,y) = c(xy)$;
- les (y,x) avec $x \in X$, $y \in Y$ et $xy \in M$, avec un poids $\lambda(y,x) = -c(xy)$.

❑ 24 Montrer que l'on peut ramener la recherche d'un chemin augmentant de coût minimal dans M à la recherche d'un chemin de poids minimal dans G_M.

❑ 25 Justifier que le graphe G_{M_k} associé au couplage obtenu après k étapes de l'algorithme 1 ne contient pas de cycle de poids négatif.

II.2 Implémentation

On représente un graphe biparti $G = (X \sqcup Y, E, c)$ par la structure suivante :

```
struct Bipartite {
    int n;
    int p;
    double **adj;
};

typedef struct Bipartite bipartite_t;
```

- `n` donne $|X|$, et les sommets de X sont numérotés de 0 à $n-1$;
- `p` donne $|Y|$, et les sommets de Y sont numérotés de n à $n+p-1$;
- `adj` est la matrice d'adjacence pondérée de G, de taille $(n+p, n+p)$. G étant biparti, on a nécessairement `adj[i][j]` qui vaut `INFINITY` si `i` et `j` sont tous les deux strictement inférieurs à n, ou tous les deux supérieurs ou égaux à n.
- On représentera un couplage d'un graphe de taille $n+p$ par un tableau `m` de $n+p$ entiers, où `m[i]` vaudra `j` si le sommet `i` est apparié au sommet `j`, et -1 si `i` est libre.
- Un chemin $v_0, \dots, v_l$ de longueur `l` sera simplement représenté par un tableau de `l`$+1$ entiers.

❏ 26 Écrire une fonction `build_oriented` qui prend en entrée un graphe biparti G et un couplage M et renvoie le graphe G_M sous forme d'une matrice d'adjacence pondérée. Si G est de taille $n+p$, on renverra donc une matrice de taille $(n+p+2, n+p+2)$, où le sommet d'indice $n+p$ correspondra à s et celui d'indice $n+p+1$ à t.

```
double **build_oriented(bipartite_t *g, int *matching);
```

❏ 27 Écrire une fonction `augment` qui prend en entrée un couplage `matching`, un chemin `path` supposé augmentant pour ce couplage et un entier `len` indiquant la longueur de ce chemin, et modifiant le couplage en effectuant l'opération $M \leftarrow M \oplus P$.

Remarque

Attention, `len` indique la longueur du chemin, qui diffère de celle du tableau `path`.

```
void augment(int *matching, int *path, int len);
```

On considère la fonction `shortest_augmenting_path` suivante qui prend en entrée un graphe G, un couplage M, et un pointeur `len` vers un entier et qui doit :

- renvoyer un chemin augmentant pour M de coût minimal, sous la forme d'un pointeur `p` vers un bloc alloué d'entiers ;
- modifier la valeur pointée par `len` pour qu'elle indique le nombre d'arêtes du chemin renvoyé ;
- dans le cas où aucun chemin augmentant n'existe, on renvoyer un pointeur nul et on fixera la valeur pointée par `len` à zéro.

```c
int *shortest_augmenting_path(bipartite_t *g, int *matching, int *len) {
    int N = g->n + g->p;
    double **G_orient = build_oriented(g, matching);
    int **from = floyd_warshall(G_orient, N);
    int s = N;
    int t = N + 1;

    *len = 0;
    int path = NULL;
    if (from[s][t] != -1) {
        path = malloc(N * sizeof(int));
        int i = 0;
        int v = from[s][t];
        while (v != s)
            path[i] = v;
            v = from[s][v];
        }
        *len = i - 1; // nombre de sommets moins un
    }

    free_double_matrix(G_orient, N + 2);
    return path;
}
```

On remarquera que cette fonction alloue un tableau potentiellement trop grand pour le chemin, mais ceci n'est pas un problème.

❏ 28 Malheureusement cette fonction comporte tout de même 5 erreurs (problème d'indice, problème de type, oubli de la libération de la mémoire, oubli d'incrément et une erreur de syntaxe). Identifier précisément les 5 erreurs.

❏ 29 Écrire une fonction `compute_optimal_matching` qui prend en entrée un graphe biparti G et renvoie un couplage optimal de G.

```c
int *compute_optimal_matching(bipartite_t *g);
```

❏ 30 Déterminer la complexité de la fonction `compute_optimal_matching`.

III Un problème d'ordonnancement

- On considère un ensemble de *tâches* $T = \{t_1, \dots, t_n\}$ qui doivent toutes être accomplies, et un ensemble de *machines* $W = \{w_1, \dots, w_m\}$ susceptibles de les accomplir.
- Chaque tâche t nécessite un certain nombre d'unités de temps pour être réalisée, et ce nombre dépend de la machine w qui la réalise : on le note $p(t, w) \in \mathbb{N}^*$.
- Une machine ne peut accomplir qu'une seule tâche à la fois : si la machine w commence à traiter la tâche t à l'instant zéro, elle ne pourra commencer à traiter une autre tâche qu'à l'instant $p(t, w)$. Si elle traite ensuite la tâche t', elle la terminera à l'instant $p(t, w) + p(t', w)$.
- *Ordonnancer* les tâches consiste à choisir une liste ordonnée $L(w)$ de tâches pour chaque machine w, de manière à avoir $T = L(w_1) \sqcup \cdots \sqcup L(w_m)$. La liste $L(w)$ indique les tâches traitées par la machine w, et l'ordre de traitement de ces tâches.

- Étant donné un ordonnancement, on définit $F(t)$ comme l'instant de fin de traitement de la tâche t.
- On cherche à trouver un ordonnancement qui minimise la somme $C = \sum_{i=1}^{n} F(t_i)$ des instants de fin de traitement.

Un exemple On considère deux machines, trois tâches, et des durées $p(t_i, w_j)$ données par le tableau ci-dessous :

	t_1	t_2	t_3
w_1	4	5	7
w_2	8	6	2

Voici deux ordonnancements possibles, parmi d'autres :

- $L(w_1) = \emptyset$ et $L(w_2) = 2, 3, 1$ a un coût total de $6 + 8 + 16 = 30$;
- $L(w_1) = 1, 2$ et $L(w_2) = 3$ a un coût total de $(4 + 9) + 2 = 15$;

❑ 31 Donner un ordonancement de coût total 14.

❑ 32 Étant donné un ordonnancement et une tâche t, on note :

- $w(t)$ la machine sur laquelle t est programmée (*i.e* $t \in L_{w(t)}$) ;
- $g(t)$ le nombre de tâches programmée après t, au sens large, sur la machine $w(t)$. On a donc $g(t) = 1$ si t est la dernière tâche à être exécutée sur la machine $w(t)$.

Montrer que $C = \sum_{t \in T} g(t)p(t, w(t))$

❑ 33 Expliquer alors comment ramener ce problème d'ordonnancement optimal à une recherche de couplage optimal dans un graphe biparti pondéré.

❑ 34 Quelle complexité obtient-on pour la résolution de ce problème d'ordonnancement ?