DS n° 02 — 2h

On veillera à présenter *très clairement* sa copie, on attachera un soin particulier à la rédaction et **on encadrera les réponses**. Des points pourront être accordés à la présentation, à la lisibilité et au soin accordé à la rédaction. Il est impératif d'utiliser un brouillon. Les programmes seront rédigés clairement et proprement, en mettant en valeur l'indentation et en choisissant des noms de variables explicites. **Les commentaires des programmes doivent dans tous les cas être dans une autre couleur que le programme lui-même**. Il est impératif de respecter l'indentation usuelle des fonctions OCaml et C.

Lorsque l'on pose des hypothèses sur les arguments, il n'est pas nécessaire dans vos fonctions de le vérifier ni de prévoir le comportement si l'hypothèse est violée. L'introduction et l'utilisation de fonctions auxiliaires est autorisée, et même encouragée, d'autant plus si cela améliore la lisibilité et la compréhension. Les fonctions auxiliaires **doivent être précédées de leur type (en OCaml) et commentées**.

Le sujet est composé de deux parties indépendantes que vous pouvez traiter dans l'ordre de votre choix.

I Classes sylvestres

Cette partie est à traiter en utilisant exclusivement le langage OCaml.

Il existe plusieurs manières d'insérer les mêmes éléments dans un arbre binaire de recherche initialement vide pour obtenir un même arbre. Étant donné un arbre binaire de recherche T, sa classe sylvestre est l'ensemble des mots qui donnent l'arbre T après insertion dans l'arbre vide. L'objectif de ce problème est d'étudier quelques propriétés de cet ensemble.

I.1 Préliminaires

Dans tout le problème on considère un alphabet Σ totalement ordonné.

On définit un *arbre binaire* étiqueté par les éléments de Σ de manière inductive :

- l'arbre vide, noté $\circ$, est un arbre binaire ;
- si $r \in \Sigma$, T_g et T_d sont des arbres binaires, alors le triplet $T = (T_g, r, T_d)$ est un arbre binaire. Les éléments r, T_g et T_d sont appelés respectivement *racine*, *sous-arbre gauche* et *sous-arbre droit* de T.

On définit un *arbre binaire de recherche* (abrégé en ABR) de manière inductive :

- l'arbre vide est un ABR ;
- si $r \in \Sigma$, T_g et T_d sont des ABR tels que toute valeur apparaissant dans T_g est **strictement inférieure** à r, et toute valeur apparaissant dans T_d est **supérieure ou égale** à r, alors le triplet $T = (T_g, r, T_d)$ est un ABR.

On définit l'emphinsertion d'un élément $a \in \Sigma$ dans un arbre binaire T comme l'opération notée $T \leftarrow a$ définie inductivement par :

- si $T = \circ$, alors $T \leftarrow a = (\circ, a, \circ)$;
- si $T = (T_g, r, T_g)$:
  - ⋆ si $r \leq a$, alors $T \leftarrow a = (T_g, r, T_d \leftarrow a)$;
  - ⋆ si $r > a$, alors $T \leftarrow a = (T_g \leftarrow a, r, T_d)$.

On définit alors inductivement l'insertion d'un mot $u \in \Sigma^*$ dans un arbre binaire T, noté également $T \leftarrow u$, comme suit :

- si $u = \varepsilon$, alors $T \leftarrow u = T$;
- si $u = av$, avec $a \in \Sigma$ et $v \in \Sigma^*$ alors $T \leftarrow u = (T \leftarrow a) \leftarrow v$.

Dans le cas où T est un ABR et $u \in \Sigma^*$, on admet que $T \leftarrow u$ est encore un ABR.

Pour T un ABR, on définit la *classe sylvestre* de T, notée $\mathsf{Syl}(T)$, comme l'ensemble des mots u vérifiant $\circ \leftarrow u = T$. Remarquons que l'on a en particulier $\mathsf{Syl}(\circ) = \{\varepsilon\}$.

Par exemple, si $T = ((\circ, a, \circ), b, (\circ, c, \circ))$, alors $\mathsf{Syl}(T) = \{bac, bca\}$.

On utilise la représentation usuelle des arbres. Par exemple, la figure 1 représente un arbre T_1 de taille 4.

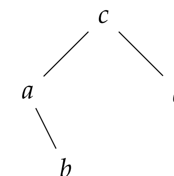


Figure 1 – Représentation de l'arbre $T_1 = ((\circ, a, (\circ, b, \circ)), c, (\circ, d, \circ))$.

Q 1) Représenter graphiquement, sans justification, l'arbre $\circ \leftarrow u$, où le mot u est le mot « banane », l'ordre de comparaison des lettres étant l'ordre alphabétique.

Q 2) Donner, sans justification, la classe sylvestre de l'arbre T_1 représenté figure 1.

Dans la suite du problème, les lettres de Σ sont représentées en OCAML par des objets de type `char` et les mots sur l'alphabet Σ par des listes de lettres.

```
OCaml
type lettre = char
type mot = lettre list
```

Par exemple, le mot « banane » sera représenté par la liste `['b'; 'a'; 'n'; 'a'; 'n'; 'e']`.

On représente les arbres binaires étiquetés par des lettres par le type suivant :

```
OCaml
type arbre =
  | V
  | N of arbre * lettre * arbre
```

Ainsi, l'arbre T_1 représenté graphiquement figure 1 pourra s'écrire en OCAML :

```
OCaml
let t1 = N (N (V, 'a', N (V, 'b', V)), 'c', N (V, 'd', V))
```

Q 3) Écrire une fonction `insertion_lettre : lettre -> arbre -> arbre` qui prend en argument une lettre `a` et un arbre `t` et qui renvoie `t` $\leftarrow$ `a`.

Q 4) En déduire une fonction `insertion_mot : mot -> arbre -> arbre` qui prend en argument un mot `u` et un arbre `t` et renvoie `t` $\leftarrow$ `u`.

Pour T un arbre binaire, on définit le mot appelé la *lecture préfixe* de T, notée $\mathsf{pref}(T)$, de manière récursive par :

- $\mathsf{pref}(\circ) = \varepsilon$;
- si $T = (T_g, r, T_d)$, alors $\mathsf{pref}(T) = r \cdot \mathsf{pref}(T_g) \cdot \mathsf{pref}(T_d)$.

Q 5) Donner, sans justification, $\mathsf{pref}(T_2)$, où T_2 est l'arbre représenté à la figure 2.

Q 6) Soit T un ABR. Montrer *très soigneusement* que $T = \circ \leftarrow \mathsf{pref}(T)$.

On a donc $\mathsf{pref}(T) \in \mathsf{Syl}(T)$ pour tout ABR T.

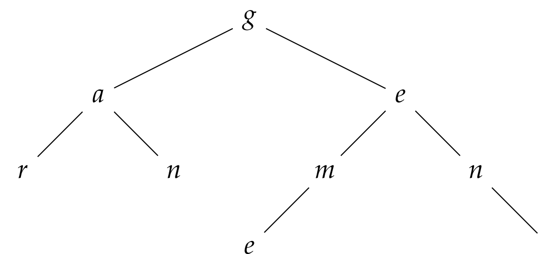


FIGURE 2 – Représentation de l'arbre T_2.

I.2 Contruction des classes sylvestres

Pour construire des classes sylvestres, on définit le *mélange* de deux mots $u, v \in \Sigma^*$, noté $u \bowtie v$, de manière inductive par :

- si $v = \varepsilon$, alors $u \bowtie v = \{u\}$;
- de même, si $u = \varepsilon$, alors $u \bowtie v = \{v\}$;
- si $u = ax$ et $v = by$, avec $a, b \in \Sigma$ deux lettres et $x, y \in \Sigma^*$ deux mots, alors :

$$u \bowtie v = \{aw \mid w \in x \bowtie v\} \cup \{bw \mid w \in u \bowtie y\}$$

Q 7) Donner, sans justification, tous les éléments de $abba \bowtie ba$.

Q 8) Écrire une fonction `ajout_lettre : lettre -> mot list -> mot list` telle que si `a` est une lettre et `lst` une liste de mots, alors `ajout_lettre a lst` renvoie une liste contenant les mots de `lst` auxquels on a rajouté la lettre `a` en début de mot.

Q 9) Écrire une fonction `melange : mot -> mot -> mot list` telle que `melange u v` renvoie $u \bowtie v$. On autorisera cette fonction à renvoyer des listes contenant des doublons que l'on ne cherchera pas à éliminer.

Si L et M sont des ensembles de mots, le *mélange* de L et M, noté $L \bowtie M$, est défini par :

$$L \bowtie M = \bigcup_{u \in L, v \in M} u \bowtie v$$

Notons que si l'un des deux ensembles est vide, alors $L \bowtie M = \varnothing$.

Q 10) Écrire une fonction `melange_ens` de type `mot list -> mot list -> mot list` telle que si `l` et `m` sont des listes de mots correspondant à des ensembles L et M, alors `melange_ens l m` renvoie $L \bowtie M$.

On admet le résultat suivant : si $T = (T_g, r, T_d)$ est un ABR, alors :

$$\mathsf{Syl}(T) = \{rw \mid w \in \mathsf{Syl}(T_g) \bowtie \mathsf{Syl}(T_d)\}$$

Q 11) Écrire une fonction `sylvestre : arbre -> mot list` telle que si `t` est un arbre, alors `sylvestre t` renvoie Syl(`t`).

I.3 Caractérisation des classes sylvestres

Soient $u, v \in \Sigma^*$. On dit que u est *S-adjacent* à v (ou que u et v sont S-adjacents) s'il existe trois lettres $a, b, c \in \Sigma$ avec $a < b \leq c$ et trois mots $x, y, z \in \Sigma^*$ vérifiant :

$$(u = xbyacz \text{ et } v = xbycaz) \quad \text{ou} \quad (u = xbycaz \text{ et } v = xbyacz)$$

On dit que u est *S-équivalent* à v (ou que u et v sont S-équivalents), noté $u \sim_S v$, s'il existe $n \in \mathbb{N}$ et des mots $w_0, w_1, \ldots, w_n \in \Sigma^*$ vérifiant :

$$u = w_0, v = w_n \text{ et } \forall i \in [\![0, n-1]\!], w_i \text{ est S-adjacent à } w_{i+1}$$

Notons qu'ici, les w_i désignent bien des mots et non des lettres.

Q 12) Montrer que $\sim_S$ est une relation d'équivalence, c'est-à-dire qu'elle est réflexive, symétrique et transitive.

Q 13) Soient $a < b \leq c$ trois lettres de Σ. Montrer que pour tout ABR T contenant au moins la lettre b, $T \leftarrow ac = T \leftarrow ca$.

Q 14) Montrer que si u et v sont S-équivalents, alors $\circ \leftarrow u = \circ \leftarrow v$.

On peut montrer que deux mots sont S-équivalents si et seulement s'ils sont des éléments d'une même classe sylvestre. Les classes sylvestres sont donc les classes d'équivalences pour la relation d'équivalence S.

II Circuits eulériens

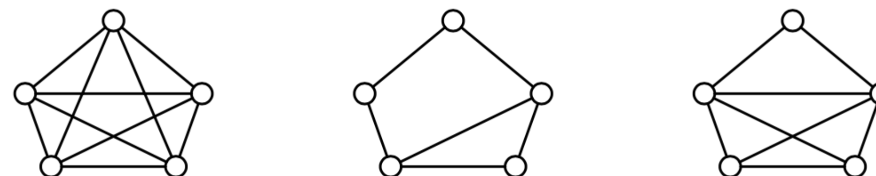
Cette partie est à traiter en utilisant exclusivement le langage C.

Dans ce problème, on s'intéresse uniquement à des graphes non orientés *connexes* contenant au moins un sommet. On suppose également qu'il n'y a pas de boucle, c'est-à-dire d'arête ayant pour source et destination un même sommet.

Dans ce problème un *cycle* est un chemin fermé (qui part d'un sommet et revient à ce dernier) de longueur non nulle qui ne passe pas deux fois par la même arête (mais peut passer plusieurs fois par un même sommet). On remarque qu'un cycle est de longueur au moins trois.

Un *cycle eulérien* est un cycle qui emprunte *chaque arête du graphe exactement une fois*. Remarquons qu'un tel chemin peut passer plusieurs fois par un même sommet.

Q 15) Pour chacun des trois graphes suivants, indiquer sans justifier s'il possède ou non un cycle eulérien.



Q 16) Montrer soigneusement que si un graphe admet un cycle eulérien alors tous ses sommets sont de degré pair.

Dans la suite de ce problème, on va démontrer que la réciproque est vraie, c'est-à-dire qu'un graphe connexe dont tous les sommets sont de degré pair admet un cycle eulérien. On va le faire en programmant un algorithme qui construit un cycle eulérien.

Pour construire un cycle eulérien, on va représenter toute arête $a - b$ du graphe par deux arcs orientés inverses ($a \to b$ et $b \to a$) en utilisant la structure `edge` donnée ci-dessous.

C
```c
typedef struct Edge {
  int src, dst;
  struct Edge *prev, *next;
} edge;
```

Une valeur de type `edge` représente un arc du graphe entre les sommets `src` et `dst`. Les deux autres champs `prev` et `next` seront utilisés plus tard et permettront de chaîner les arcs dans une liste circulaire doublement chaînée, afin de représenter un cycle dans le graphe. Même si le graphe est non orienté, on choisit donc de voir une structure `edge` comme un arc orienté allant de `src` à `dst`, l'arc inverse étant aussi présent dans le graphe.

On identifie les sommets d'un graphe avec les entiers $0, 1, \dots, N-1$ où N est une constante globale définie une fois pour toutes. Le graphe est donné dans une variable globale `graph`, sous la forme d'une matrice d'adjacence, dans laquelle on représente toute arête du graphe non orienté par deux arcs orientés inverses. S'il n'y a pas d'arc entre les sommets `i` et `j`, alors `graph[i][j]` et `graph [j][i]` valent `NULL`. S'il y a un arc entre les sommets `i` et `j`, alors `graph[i][j]` pointe vers une structure `e` telle que `e.src == i` et `e.dst == j` et, inversement, `graph[j][i]` pointe vers une structure `e` telle que `e.src == j` et `e.dst == i`. Les champs `next` et `prev` de ces deux structures sont `NULL`. Autrement dit, les arcs ne sont pas connectés entre eux initialement. Au fur et à mesure de la construction du cycle eulérien, ces arcs seront retirés du graphe et connectés entre eux pour former un cycle.

C
```c
#define N ... // On suppose la taille du graphe N définie ici

edge *graph[N][N];  // Matrice d'adjacence

// Indique s'il existe au moins un arc sortant du sommet v
bool has_edge(int v) {
  for (int j = 0; j < N; j++)
    if (graph[v][j] != NULL)
      return true;
  return false;
}

// Renvoie un pointeur sur arc sortant de v.
// Cet arc (et l'arc inverse) sont supprimés.
edge *any_edge_from(int v) {
  for (int j = 0; j < N; j++) {
    if (graph[v][j] == NULL) continue;
    edge *e = graph[v][j];
    graph[v][j] = NULL;
    graph[j][v] = NULL;
    return e;
  }
  assert(false);
}
```

On suppose disposer du programme précédent pour représenter le graphe. La fonction `has_edges` détermine si un sommet donné possède au moins un arc. Étant donné un sommet `v` qui possède au moins un arc, la fonction `any_edge_from` renvoie un pointeur sur un arc `e` tel que `e.src == v`, après l'avoir supprimé du graphe (ainsi que l'arc inverse).

On commence par un simple test du fait que le critère est vérifié.

Q 17) Écrire une fonction de prototype **`bool is_eulerian(void)`** qui détermine si le graphe possède uniquement des sommets de degré pair. Votre fonction devra laisser le graphe inchangé.

Q 18) Soit un graphe connexe possédant au moins deux sommets et soit v_0 un sommet de ce graphe. Justifier que v_0 est incident à au moins une arête.

Q 19) Soit un graphe connexe possédant au moins deux sommets, dont tous les sommets ont un degré pair. Soit un sommet v_0 dans ce graphe. Montrer que le graphe possède un cycle (pas nécéssairement eulérien) partant de v_0. *Indication : on pourra raisonner par l'absurde et construire pour tout $k \in \mathbb{N}$ un chemin $v_0 - v_1 - \cdots - v_k$ tel que chaque arête n'est prise qu'une fois et tel que $v_i \neq v_0$ pour tout $i > 0$.*

La fonction `connect` suivante permet de connecter deux arcs pour construire un cycle avec les arêtes que l'on retire du graphe.

C
```c
// Relie l'arc x à l'arc y, en supposant x->dst == y->src
void connect(edge *x, edge *y) {
  assert(x->dst == y->src);
  x->next = y;
  y->prev = x;
}
```

Q 20) On suppose que le graphe contient encore des arcs sortant d'un sommet `start` et que tous les sommets sont de degré pair. Écrire une fonction de prototype `edge *round_trip(int start)` qui construit un cycle (pas nécessairement eulérien) à partir du sommet `start` en utilisant des arcs encore présents dans le graphe et les retire au fur et à mesure (grâce à la fonction `any_edge_from`). L'arc renvoyé est l'arc de ce cycle dont le champ `src` est `start`. Cette fonction ne pourra pas échouer, en vertu de la question précédente. On prendra soin de bien mettre à jour les champs `next` et `prev` des arcs utilisés, en appelant la fonction `connect` fournie.

Q 21) Écrire une fonction de prototype `edge *find_vertex_with_edges(edge *c)` qui reçoit en argument un cycle et cherche le long de ce cycle un sommet qui possède encore des arcs. S'il existe un tel sommet `v`, cette fonction renvoie un pointeur sur un arc `e` de ce cycle pour lequel `e.src == v`. Si en revanche il n'existe aucun sommet le long du cycle possédant encore des arcs, cette fonction renvoie `NULL`.

Algorithme de Hierholzer Pour construire un cycle eulérien, on peut procéder ainsi. On commence par construire un cycle arbitraire, avec la fonction `round_trip` en partant d'un sommet initial quelconque. Si tous les arcs ont été utilisés, on a terminé. Sinon, on prend un sommet `v` sur le cycle qui possède encore des arcs. C'est possible, car le graphe est connexe. À partir de v, on construit un nouveau cycle, toujours avec la fonction `round_trip`. Puis on joint les deux cycles pour n'en former qu'un seul. Et ainsi de suite jusqu'à épuisement des arcs. Pour joindre deux cycles, on utilise la fonction `join` ci-dessous.

C

```c
// Joint les deux cycles c1 et c2, en supposant que
// c1->src == c2->src
void join(edge *c1, edge *c2) {
  assert(c1->src == c2->src);
  edge *p = c1->prev;
  edge *q = c2->prev;
  connect(p, c2);
  connect(q, c1);
}
```

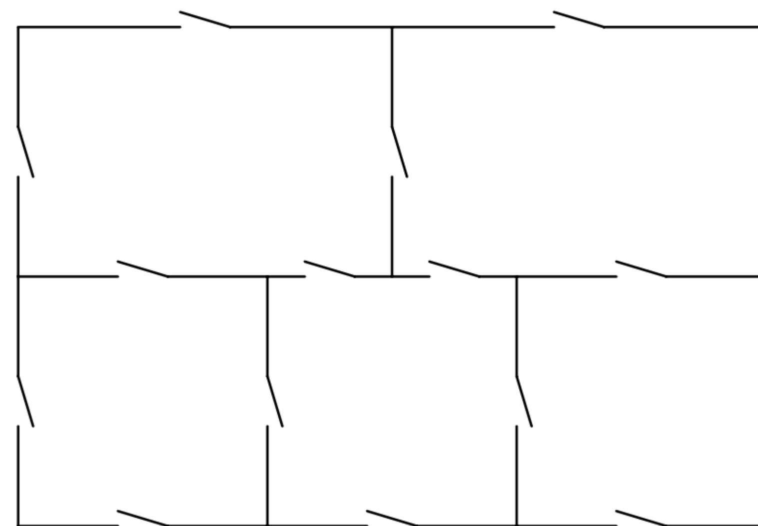
Q 22) Écrire une fonction `edge *eulerian_cycle(int start)` qui reçoit en argument un sommet initial start et construit et renvoie un cycle eulérien en suivant l'algorithme de Hierholzer.

Q 23) Donner la complexité dans le pire des cas de l'algorithme de Hierholzer tel que nous l'avons écrit, en fonction du nombre N de sommets et du nombre E d'arcs du graphe.

Un *chemin eulérien* dans un graphe non orienté est un chemin qui emprunte chaque arête une et une seule fois, sans nécessairement revenir au point de départ.

Q 24) Montrer qu'un graphe admet un chemin eulérien si et seulement s'il possède 0 ou 2 sommets de degré impair. Pour la réciproque, on s'attachera à décrire un algorithme construisant le chemin (mais on ne demande pas d'en écrire le code).

Q 25) Est-il possible de tracer un chemin continu qui traverse les cinq pièces suivantes en empruntant chaque porte une et une seule fois ? Le chemin n'a pas besoin d'être cyclique. Si oui, exhiber un tel chemin ; si non, justifier.



— Fin du sujet —