ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES
CONCOURS D'ADMISSION 2023

JEUDI 20 AVRIL 2023
14h00 - 18h00
FILIERE MPI - EPREUVE n° 7
INFORMATIQUE C (XULSR)

Durée : 4 heures
L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Cette composition ne concerne qu'une partie des candidats de la filière MP, les autres candidats effectuant simultanément la composition de Physique et Sciences de l'Ingénieur. Pour la filière MP, il y a donc deux enveloppes de Sujets pour cette séance.

Ordonnabilité des espaces métriques

Le sujet comporte 12 pages, numérotés de 1 à 12

Début de l'épreuve

Dans ce sujet, on s'intéresse au problème d'ordonner les éléments d'un espace métrique de sorte que deux éléments successifs soient à distance bornée.

Ce sujet est constitué de quatre parties. La première partie est formée de préliminaires utiles dans le reste du sujet, avec des questions de programmation en C et OCaml. La deuxième partie propose une implémentation en C d'une structure de données utilisée notamment dans la troisième partie. Cette troisième partie considère les graphes, munis de la distance de plus court chemin, comme espace métrique, avec implémentations en C. La quatrième partie, enfin, indépendante des deux précédentes, considère l'ordonnabilité des langages réguliers pour une certaine distance d'édition ; cette partie contient de la programmation en OCaml.

Dans les questions de programmation en C ou OCaml, on n'utilisera pas de fonctions qui ne sont pas incluses dans al bibliothèque standard du langage. Pour les codes en C, on pourra supposer que les en-têtes `<stdbool.h>`, `<stdio.h>`, `<stdlib.b>` et `<assert.h>` ont été inclus.

Espaces métriques et d-ordres

Dans ce sujet, on s'intéresse à des collections de données discrètes que l'on cherche à explorer ou engendrer de proche en proche. On modélise ce cadre général à l'aide des notions d'*espace métrique*, de *d-suite* et de *d-ordre* :

Définition 1 (espace métrique). Un *espace métrique* $\mathcal{M} = (X, \delta)$ est constitué d'en ensemble dénombrable X dont les éléments sont appelés *points* et d'une *distance* sur X, c'est-à-dire une application $\delta : X \times X \to \mathbb{N}$ satisfaisant :

Symétrie. Pour tout $(x, y) \in X^2, \delta(x, y) = \delta(y, x)$.

Séparation. Pour tout $(x, y) \in X^2, \delta(x, y) = 0$ si et seulement si $x = y$.

Inégalité triangulaire. Pour tout $(x, y, z) \in X^3, \delta(x, y) \leq \delta(x, z) + \delta(z, y)$.

Pour un espace métrique $\mathcal{M} = (X, \delta)$ et $X' \subseteq X$, on note $\mathcal{M}[X']$ le couple $(X', \delta_{|X'})$, où $\delta_{|X'}$ dénote la restriction de δ au domaine $X' \times X'$. on observe que $\mathcal{M}[X']$ est encore un espace métrique, que l'on appellera *sous-espace* de $\mathcal{M}$.

Définition 2 (d-suite). Pour $d \in \mathbb{N}^*$, une *d-suite* s dans un espace métrique $\mathcal{M} = (X, \delta)$ est ue suite (finie ou infinie dénombrable) $s = x_1, x_2, ...$ de points de $\mathcal{M}$ telle que :

- s ne contient pas de doublons, c'est-à-dire que pour tout x_i, x_j de la suite, si $i \neq j$ alors $x_i \neq x_j$;
- deux points consécutifs de la suite snt à distance au plus d ; formellement, pour tout x_i, x_{i+1} de la suite, on a $\delta(x_i, x_{i+1}) \leq d$.

On dit que s commence en x_1 et, dans le cas où s est finie et a n éléments, que s termine en x_n.

Définition 3 (d-ordre, d-ordonnable). Pour $d \in \mathbb{N}^*$, un *d-ordre* de $\mathcal{M}$ est une d-suite dans $\mathcal{M}$ contenant tous les points de $\mathcal{M}$. L'espace $\mathcal{M}$ est dit *d-ordonnable* lorsqu'il existe un d-ordre de $\mathcal{M}$. On dit que $\mathcal{M}$ est *ordonnable* lorsque $\mathcal{M}$ est d-ordonnable pour un certain d.

Distances d'édition sur les mots

On fixe dans tous le sujet l'*alphabet* $\Sigma := \{a, b\}$ ayant pour seules lettres a et b. Un mot w est une suite finie de lettres $w = \alpha_1 ... \alpha_n$. La longueur de w, notée $|w|$, est n. On note ε le mot vide, de longueur nulle. On note Σ^* l'ensemble des mots sur Σ. Un *langage* est un sous-ensemble de Σ^*.

Un espace métrique d'intérêt, sur l'ensemble des mots d'un langage, est donné par deux distances d'édition sur les mots : la distance *push-pop* et la distance *push-pop-droite*, que l'on définit ci-après.

Définition 4 (distance push-pop). La *distance push-pop*, dénoté δ_{pp} est définie de la manière suivante : pour $w, w' \in \Sigma^*$, $\delta_{pp}(w, w,')$ est le nombre minimal d'*opérations* nécessaires pour passer de w à w', où les opérations autorisées sont :

— pour un mot w, *insérer la lettre* $\alpha \in \Sigma$ *à la fin*, ce qui donne le mot $w\alpha$;

— pour un mot w, *insérer la lettre* $\alpha \in \Sigma$ *au début*, ce qui donne le mot αw ;

— pour un mot de la forme $w\alpha$ avec $\alpha \in \Sigma$, *supprimer la dernière lettre*, ce qui donne le mot w ;

— pour un mot de la forme αw avec $\alpha \in \Sigma$, *supprimer la première lettre*, ce qui donne le mot w.

Exemple 1. Les mots à distance push-pop du mot *aab* sont *aa* (on a supprimé la dernière lettre), *aaba* (on a ajouté un *a* à la fin), *aabb* (on a ajouté un *b* à la fin), *ab* (on a supprimé la première lettre), *aaab* (on a ajouté un *a* au début) et *baab* (on a ajouté un *b* au début).

Définition 5 (distance push-pop-droite). La distance *push-pop-droite*, dénotée par δ_{ppr}, est définie de la même manière que δ_{pp} mais seules les insertions et suppressions à la fin du mot sont autorisées.

Exemple 2. Les mot qui sont à distance push-pop-droite 1 du mot *aab* sont *aa* (on a supprimé la dernière lettre), *aaba* (on a ajouté un *a* à la fin) et *aabb* (on a ajouté un *b* à la fin).

Algorithmes d'énumération push-pop et push-pop-droite

Lorsqu'on travaillera sur les mots, on considérera parfois des programmes produisant une suite (potentiellement infinie) de mots de Σ^*, en utilisant les instructions spéciales suivantes : `popL()`, `popR()` ; `pushL(`α`)` et `pushR(`α`)` pour $\alpha \in \Sigma$; et `output()`.

Le programme dispose, comme état interne, d'une liste L d'éléments de Σ, qui est interprétée comme un mot de Σ^*. La liste L est initialement vide et représente donc le mot vide. Voici ce qu'il se passe lorsque le programme utilise les instructions spéciales :

— `popL()` a pour effet de supprimer la première lettre de la liste (et ne peut être appelée que si la liste est non vide) ;

— `popR()` a pour effet de supprimer la dernière lettre de la liste (et ne peut être appelée que si la liste est non vide) ;

— `pushL(`α`)` a pour effet d'ajouter la lettre $\alpha \in \Sigma$ en début de liste ;

— `pushR(`α`)` a pour effet d'ajouter la lettre $\alpha \in \Sigma$ en fin de liste ;

— `output()` *produit* le mot étant actuellement représenté par la liste (par exemple, il l'affiche en sortie du programme).

On souligne que la liste L n'est accessible par le programme que via ces instructions spéciales. De plus, on fait l'hypothèse que chacune des instructions `popL`, `popR`, `pushL` et `pushR` a une complexité en $O(1)$.

Un tel programme *produit* alors la suite $w_1, w_2, ...$, où w_1 est le premier mot produit par le programme lors de son exécution, w_2 le secont et ainsi de suite.

On appellera un tel programme un *programme push-pop*. De manière similaire, un programme *push-pop-droite* est un programme push-pop qui n'utilise par les instructions `popL()` ni `pushL(`α`)` pour $\alpha \in \Sigma$.

Exemple 3. On suppose que les fonctions C `pushL`, `pushR`, `popL`, `popR`, `output` ont été prédéfinies et manipulent toutes une même variables représentant la liste. Le programme C suivant est une programme push-pop-droite qui produit la suite de mots $w_1, w_2, ...$ où w_i est $(aa)^{i-1}b$:

```
int main(void)
{
    pushR('b') ; output() ;
    while (true) {
        popR() ; pushR('a') ; pushR('a') ; pushR('b') ; output() ;
    }
}
```

On remarque que ce programme ne se termine jamais.

En OCaml, on suppose qu'on dispose d'un type `lettre` défini par :

```
type lettre = A | B
```

On représente alors les mots de Σ^* par des listes OCaml `lettre list`, dont la tête correspond à la première lettre du mot. Par exemple, le mot *aab* est représenté par la liste `[A;A;B]`. On suppose avoir accès en OCaml à des fonctions de type `pushR : lettre -> unit`, `pushL : lettre -> unit`, `popR : unit -> unit`, `popL : unit -> unit`, ainsi que `output : unit -> unit`, qui représentent les instructions spéciales.

Exemple 4. La programme OCaml push-pop suivant produit la même suite que le programme C push-pop-droite de l'exemple 3 :

```
let main () =
    pushR B ; output () ;
    while true do
        pushL A ; pushL A ; output () ;
    done;
```

Partie I. Préliminaires

Question 1.1. Soit $\mathcal{M} = (X, \delta)$ un espace métrique tel que X est un ensemble fini. Montrer que $\mathcal{M}$ est ordonnable.

Question 1.2. Écrire une fonction OCaml

```
delta_ppr : lettre list -> lettre list -> int
```

prenant en entrée deux liste `l1`, `l2` représentent des mots w_1, w_2 et renvoyant $\delta_{ppr}(w_1, w_2)$, la distance push-pop-droite entre w_1 et w_2. On attend de cette fonction que sa complexité soit linéaire en $|w_1| + |w_2|$.

Question 1.3. On considère la suite $s := w_1, w_2, ...$, où w_i est a^{i^2}. Écrire un programme push-pop-droite en C qui produit la suite s.

Question 1.4. Montrer que $\mathcal{M}_{pp} := (\Sigma^*, \delta_{pp})$ et $\mathcal{M}_{ppr} := (\Sigma^*, \delta_{ppr})$ sont des espaces métriques.

On s'intéresse maintenant aux sous-espaces de $\mathcal{M}_{pp}$ et $\mathcal{M}_{ppr}$, c'est-à-dire (comme indiqué dans la Définition 1) aux espaces métriques de la forme $\mathcal{M}_{pp}[L] = (L, \delta_{pp|L})$ ou $\mathcal{M}_{ppr}[L] = (L, \delta_{ppr|L})$, pour L un langage. Par exemple, la suite produit par les programmes push-pop des exemples 3 et 4 est un 4-ordre pour $\mathcal{M}_{ppr}[L]$ et un 2-ordre pour $\mathcal{M}_{pp}[L]$, où $L := (aa)^*b$.

Question 1.5. Écrire un programme push-pop en C qui produit un d-ordre pour $\mathcal{M}_{pp}[L]$, où $L := a^*b^*|b^*a^*$ et un certain $d \in \mathbb{N}$. Expliquer comment fonctionne ce programme.

Question 1.6. Écrire un programme push-pop en C qui produit un 1-ordre pour $\mathcal{M}_{pp}[L]$, où $L := a^*b^*$. [*Indice* : on peut visualiser le langage Lcomme la grille $\mathbb{N} \times \mathbb{N}$, où la position (i, j) correspond le mot $a^i b^j$] Ne pas hésiter à faire un dessin pour se faire comprendre.

Question 1.7. Soit $L := b^*|ab^*$

a. Écrire un programme push-pop en C qui produit un 1-ordre pour $\mathcal{M}_{pp}[L]$.

b. Prouver que $\mathcal{M}_{ppr}[L]$ n'est pas ordonnable.

Question 1.8. Un *chemin hamiltonien* dans un graphe non-orienté est un chemin (aussi appelé *chaîne*) qui visite tous les sommets du graphe exactement une fois. Un *graphe grille* est un graphe de la forme $G = (V, E_V)$ avec V une partie finie de $\mathbb{N} \times \mathbb{N}$ et $E_V = \{\{(i, j), (i', j')\} | (i, j), (i', j') \in V \text{ et } |i - i'| + |j - j'| = 1\}$, autrement dit, deux sommets sont connectés par une arête s'ils sont à distance 1 dans la grille $\mathbb{N} \times \mathbb{N}$.

On définit le problème de décision **HamGrille** de la manière suivante : en entrée on a un graphe grille G et la sortie est OUI si G possède un chemin hamiltonien et NON sinon. On admettra que ce problème est NP-complet.

Pour t un entier naturel non nul, on définit le problème de décision $\mathcal{P}_t$, de la façon suivant : $\mathcal{P}_t$ prend en entrée un ensemble fini de mots L et la sortie est OUI si $\mathcal{M}_{pp}[L]$ est t-ordonnable, NON sinon. On fixe maintenant un entier naturel $t \geq 1$ arbitraire. Montrer que pour cet enter t, la problème $\mathcal{P}_t$ est NP-complet.

Partie II. Implémentation en C des programmes push-pop

Dans cette partie, on souhaite réaliser une implémentation en langage C des opérations des programmes push-pop sur une structure de donnnées `liste` codant une liste doublement chaînée. Le type de la structure de données `liste` est défini comme suit :

```
typedef struct chainon_s chainon ;

struct chainon_s
{
    int val ;
    chainon *prec ;
    chainon *suiv ;
};

struct liste_s
{
    chainon *premier ;
    chainon *dernier ;
};

typedef struct liste_s liste ;
```

En particulier les lettres sont donc représentées par des entiers (on rappelle qu'en C, une variable de type `char` peut être implicitement interprétée comme une variable de type `int` sans perte d'informations) ; cela permettra de réutiliser cette structure `liste` dans d'autres contextes.

Dans l'implémentation de cette structure de données, on veillera à ce qu'un pointeur dont la valeur n'est pas `NULL` pointe toujours vers un espace mémoire valide et pas, par exemple, vers une donnée libérée.

Question 2.1. Définir et initialiser une variable globale `lg` représentant la liste chaînée globale manipulée par les programmes push-pop ; on souhaite que cette liste soit initialement vide.

Question 2.2. Programmer une fonction `est_vide` de prototype `bool est_vide(void)` renvoyant `true` si `lg` représente une liste vide, `false` sinon.

Question 2.3. Programmer des fonction `pushL` et `pushR` de prototype `void pushL(int)` et `void pushR(int)` implémentant les opérations `pushL` et `pushR` sur la liste représentée par `lg`. On utilisera une assertion pour vérifier que l'allocation dynamique de mémoire est bien réalisée. Quelle est la complexité en temps de ces deux fonctions ?

Question 2.4. Programmer des fonction `popL` et `popR` de prototype `int popL(void)` et `int popR(void)` implémentant les opérations `popL` et `popR` sur la liste représentée par `lg` (chacune renvoyant également la valeur extraite de la liste). On utilisera une assertion pour s'assurer que la liste n'est pas vide et veillera à libérer la mémoire devenue inutile. Quelle est la complexité en temps de ces deux fonctions ?

Question 2.5. Programmer des fonctions `output` et `vide_liste` de prototype respectifs `void output(void)` et `void vide_liste(void)` ; `output` affiche le mot codé par la liste de caractère sur la sortie standard (suivie d'un retour à la ligne), tandis que `vide_liste` vide la liste (retire tous ses éléments) et libère la mémoire associée. Quelle est la complexité en temps de ces deux fonctions ?

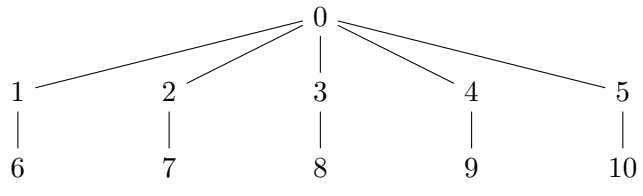
Partie III. Ordonnabilité des graphes

Dans cette partie, on s'intéresse à des espaces métriques définis à partir de graphes non-orientés connexes. Pour un graphe non-orienté connexe $G = (V, E)$ (où V est l'ensemble fini non-vide de noeuds du graphe et E sons ensemble d'arêtes), on note $\delta_G : V \times V \to \mathbb{N}$ la fonction qui à un couple de noeuds (u, v) de V associe la longueur d'un plus court chemin de u à v dans G.

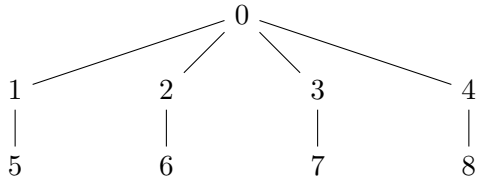
L'objectif est de montrer que les espaces de la forme $\mathcal{M}_G := (V, \delta_G)$ sont 3-ordonnables et d'étudier des algorithmes qui calculent de tels ordres. On travaillera en langage C.

Question 3.1. Soit $G = (V, E)$ un graphe non-orienté connexe. Montrer que le couple $\mathcal{M}_G := (V, \delta_G)$ est effectivement un espace métrique.

Question 3.2. On considère les graphes G_1 et G_2 représentées ci-dessous.



G_1



G_2

a. Donner un 3-ordre de $\mathcal{M}_{G_1}$ (aucune justification n'est attendue).

b. Donner un 2-ordre de $\mathcal{M}_{G_2}$ (aucune justification n'est attendue).

On souhaite calculer un arbre couvrant d'un graphe non-orienté connexe. Un *arbre* est un graphe orienté *acyclique* $T = (V_T, E_T)$ avec V_T un ensemble fini non vide de noeuds et $E_T \subseteq V_T \times V_T$ un ensemble d'arcs tel que :

— il existe un unique noeud $r \in V_T$, appelé la *racine* de T tel que pour tout $v \in V_T$, $(v, r) \notin E_T$;

— pour tout noeud $u \in V_T$ qui n'est pas la racine, il existe un unique noeud $v \in V_T$ tel que $(u, v) \in E_T$ et on dit que u est un fils de v.

On définit maintenant un *arbre couvrant* d'un graphe non-orienté conenxe $G = (V, E)$ comme un arbre $T = (V', E')$ tel que :

— $V' = V$

— pour tout $(u, v) \in E', \{u, v\} \in E$.

On va travailler avec la représentation en matrice d'adjacence des graphes : on suppose que le nombre de noeuds N est prédéfini en C comme une constante globale `N`, que les noeuds de G sont $\{0, ..., N-1\}$ et que G est représenté par un tableau bidimensionnel `bool graphe[N][N]` où `g[u][v]` et `g[v][u]` valent tous les deux `true` si $\{u, v\}$ est une arête de G, et `false` sinon.

L'arbre couvrant T sera également représenté par un tableau `bool arbre[N][N]`, où cette fois `arbre[u][v]` vaut `true` si v est un fils de u et `false` sinon.

Question 3.3. Ecrire une fonction

```
void calcule_arbre_couvrant (bool graphe[N][N], bool arbre[N][N])
```

qui remplit la structure `arbre` pour que celle-ci représente un arbre couvrant quelconque du graphe G représenté par `graphe`, avec le noeud 0 comme racine. On supposera G connexe et on ne vérifiera pas ce point dans le code. On pourra utiliser les fonctions définies dans la partie 2 pour gérer une file (ou une pile) de noeuds à traiter. Justifier brièvement la correction de votre algorithme.

On supposera par la suite que `arbre` représente effectivement un arbre couvrant comme indiqué plus haut et que le noeud 0 est la racine de celui-ci.

Question 3.4. On considère la fonction C suivante :

```
void visite(bool arbre[N][N], int noeud, bool mystere) {
    if (!mystere)
        pushR(noeud);
    for (int fils = 0; fils<N; ++fils) {
        if (arbre[noeud][fils])
            visite(arbre, fils, !mystere);
    }
    if (mystere)
        pushR(noeud);
}
```

On suppose que la liste `lg` utilisée par la fonction `pushR` est initialement vide et qu'on appelle `visite(arbre, 0, false)` sur un `arbre` de racine 0.

a. Que vaut le troisième argument `mystere` lors d'un appel récursif à `visite` avec comme deuxième argument un noeud n de l'arbre, en fonction de la position de n dans l'arbre ?

b. Que contient la liste `lg` à la fin de l'exécution de ce programme sur le graphe G_2 de la question 3.2, vu comme un arbre enraciné en 0 ?

Question 3.5. Montrer que l'appel `visite(a, 0, false)` lorsque `a` est la représentation d'un arbre couvrant d'un graphe non-orienté connexe G de racine 0, calcule (dans la liste `lg`) un 3-ordre pour $\mathcal{M}_G$.

Question 3.6. Quelle est la complexité en temps de l'appel à `visite(a, 0, false)`, en fonction du nombre N de noeuds ? Peut-on améliorer cette complexité (par exemple en changeant notre méthode de représentation des graphes) ?

Question 3.7. On observe que la fonction `visite` est une fonction récursive.

a. Quel(s) problème(s) cela pourrait-il poser ?

b. Proposer une méthode pour rendre ce calcul non récursif. On ne s'attend pas ici à du code, mais à une explication qui puisse être facilement transformée en du code.

Partie IV. Ordonnabilités des langages réguliers pour la distance push-pop-droite

Dans cette partie, on souhaite caractériser les langages réguliers ordonnables pour la distance push-pop-droite. On travaillera en OCaml.

Automates et langages réguliers. Le termes *automate* désignera systématiquement un automate fini déterministe. Formellement, un automate $A = (Q, q_{init}, F, trans)$ consiste en un ensemble fini Q d'*états*, un *état initial* q_{init}, un ensemble $F \subseteq Q$ d'*états finaux*, ainsi qu'une *fonction de transition* $trans : Q \times \Sigma \to Q \cup \{\bot\}$ où $\bot$ signifie que la transition n'est pas définie. Un *chemin* dans A d'un état $q \in Q$ à un état $q' \in Q$ est une suite finie de la forme $q_0, \alpha_1, q_2, \alpha_2, ..., \alpha_{n-1}, q_n$ où les q_i sont des états de Q et les α_i des lettres de Σ, telle que $q = q_1$, $q' = q_n$ et telle que pour tout $i \in \{1, ..., n-1\}$ on a $q_{i+1} = trans(q_i, \alpha_i)$; l'*étiquette* d'un tel chemin est alors le mot $\alpha_1 ... \alpha_{n-1}$. En particulier, il y a toujours un chemin de longueur nulle, avec étiquette ε, entre n'importe quel état et lui-même (la suite comprenant ce seul état). Le *langage accepté par* A, noté $L(A)$, est l'ensemble des mots qui étiquettes un chemin depuis q_{init} jusqu'à un état final. Un langage est *régulier* s'il est accepté par un automate ou, de manière équivalent, s'il est représenté par une expression régulière.

On rappelle qu'un automate $A = (Q, q_{init}, F, trans)$ est *émondé* si tout état q est à la fois accessible depuis l'état initial (c'est-à-dire qu'il y a un chemin depuis q_{init} à q) et co-accessible depuis un état final (c'est-à-dire qu'il y a un chemin depuis q vers un état final). Pour tout automate A, on peut calculer en temps linéaire un automate A' émondé tel que $L(A) = L(A')$.

Automates poêle. D'après la question 1.1, si L est fini alors $\mathcal{M}_{ppr}[L]$ est ordonnable. **On supposera L infini dans cette partie.** On définit dans ce qui suit la notion d'*automate poêle-à-frire* (ou *automate poêle*), puis on montre que $\mathcal{M}_{ppr}[L]$ est ordonnable si et seulement si n'importe quel automate émondé acceptant L est un automate poêle.

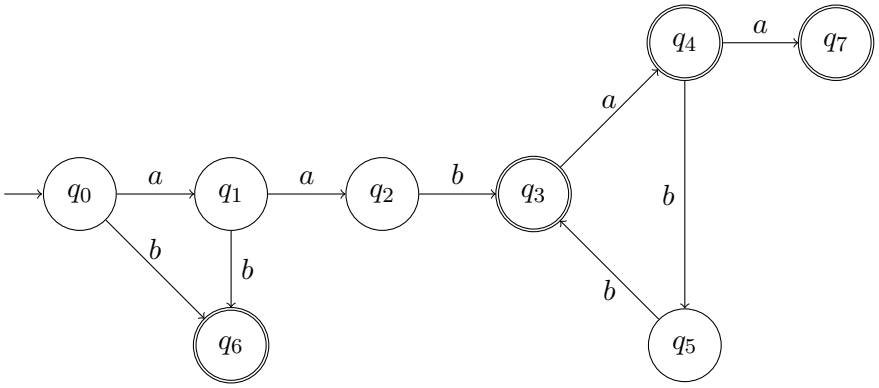
Soit $A = (Q, q_{init}, F, trans)$ un automate. Un *cycle* dans A est un chemin de longueur non nulle dans A depuis un état q vers lui-même sans répétition d'état (à part q). On observe que ce chemin est possiblement de longueur 1, dans le cas où $trans(q, \alpha) = q$ pour $\alpha \in \Sigma$. On note que si $q_1, \alpha_1, q_2, \alpha_2, ..., \alpha_{n-1}, q_1$ est un cycle, alors $q_2, \alpha_2, ..., \alpha_{n-1}, q_1, \alpha_1, q_2$ et plus généralement $q_i, \alpha_i, ..., \alpha_{n-1}, q_1, \alpha_1, ..., \alpha_{i-1}, q_i$ pour $2 \leq i \leq n-2$ sont également des cycles : on dit que ce sont les mêmes cycles *à permutation prés.*

Un *chemin strict vers un cycle* dans A est un chemin sans répétition d'état depuis l'état initial jusqu'à un état faisant partie d'un cycle tel que tous les états sauf le dernier ne font pas partie d'un cycle dans A ; formellement, c'est un chemin $q_1, \alpha_1, ..., \alpha_{n-1}, q_n$ pour $n \in \mathbb{N}$ avec $q_1 = q_{init}$ tel que q_n fait partie d'un cycle et pour $1 \leq i < n$, q_i ne fait partie d'aucun cycle. En particulier, si q_{init} fait partie d'un cycle alors le seul tel chemin est de longueur nulle.

On dit de A qu'il est *pseudo-acyclique* s'il a au plus un cycle à permutation prés. On note qu'un automate tel que $trans(q, a) = trans(q, b) = q$ n'est jamais pseudo-acyclique, car q, a, q et q, b, q sont deux cycles différents.

Un automate A est alors un *automate poêle* s'il est pseudo-acyclique et a un unique chemin strict vers un cycle.

Exemple 5. Considérons l'automate ci-dessous :



Son ensemble d'états est $\{q_0, ..., q_7\}$, son état initial q_0, ses états finaux q_3, q_4, q_6 et q_7 ; les transitions non spécifiées par des flèches sont non définies. Cet automate est clairement déterministe et il est aisé de vérifier qu'il est émondé.

L'automate est également pseudo-acyclique : il comporte en effet un unique cycle, le cycle $q_3, a, q_4, b, q_5, b, q_3$ (qu'on peut également écrire $q_4, b, q_5, b, q_3, a, q_4$ ou encore $q_5, b, q_3, a, q_4, b, q_5$). Et comme il a un unique chemin strict vers un cycle (le chemin $q_0, a, q_1, a, q_2, b, q_3$), c'est un automate poêle.

Si une transition de q_3 à q_4 étiquetée par b était ajoutée, ce ne serait plus un automate pseudo-acyclique car $q_3, a, q_4, b, q_5, b, q_3$ et $q_3, b, q_4, b, q_5, b, q_3$ seraient deux cycles différents. De même, si une transition étiquetée par a de q_2 à l'un des états du cycle était ajoutée, l'automate aurait deux chemin strict distinct vers un cycle et ne serait donc plus un automate poêle.

Question 4.1. Proposer une méthode permettant de déterminer si un automate émondé est un automate poêle, en supposant n'importe quelle représentation raisonnable des automates (on supposera l'automate émondé). On ne demande pas du code, mais on attend une explication qui puisse être facilement transformée en du code.

Lorsque A est un automate poêle, on appelle *état d'entrée* l'état qui se trouve à la fin de l'unique chemin strict vers l'unique cycle de A (cet état peut être l'état initial si celui-ci fait partie du cycle). Dans l'exemple 5, l'état q_3 est l'état d'entrée.

Question 4.2. Si A est un automate poêle, montrer que $L(A)$ peut s'écrire de la forme

$$F|uv^*F'$$

où F et F' sont des ensembles finis de mots et u, v sont des mots avec $v \neq \varepsilon$.

Question 4.3. On suppose que A est un automate poêle. Écrire un programme push-pop-droite en OCaml qui calcule un d-ordre pour $\mathcal{M}_{ppr}[L(A)]$, pour un certain $d \in \mathbb{N}$ que l'on ne cherchera pas à calculer, étant donnés les mots de u, v et ensemble F, F' de la question précédente représentés en OCaml par des variables :

```
u : lettre list
v : lettre list
f : (lettre list) list
f' : (lettre list) list
```

On a ainsi établi que lorsqu'un langage L est accepté par un automate poêle alors il est ordonnable pour la distance push-pop-droite. On montre dans la suite de cette partie que c'est en fait une condition nécessaire, dans le sens où si $\mathcal{M}_{ppr}[L]$ est ordonnable, alors n'importe quel automate émondé pour L est un automate poêle.
Pour ce faire, on considère l'arbre infini T_Σ de Σ^* défini ainsi : les noeuds de T_Σ sont les mots de Σ^*, le mot vide est la racine de T_Σ et si $w \in \Sigma^*$ alors wa et wb sont les fils de w dans T_Σ. Une *branche infinie* B dans T_Σ est une suite infinie de la forme $w_1, w_2, ...$, où $w_1 = \varepsilon$ et $w_{i+1} = w_i\alpha_i$ avec $\alpha_i \in \Sigma$ pour tout $i \in \mathbb{N}$. Pour $w \in \Sigma^*$, on note T_w le sous-arbre infini de T_Σ enraciné en w.

Pour un langage L, une *branche lourde* est une branche infinie $B = w_1, w_2, ...$ dans T_Σ telle que pour tout $i \in \mathbb{N}$, T_{w_i} contient une infinité de mots de L.

Question 4.4. Montrer que, pour n'importe quel langage L, si L est infini, alors L a (au moins) une branche lourde.

Question 4.5. Montrer que, pour n'importe quel langage L, si $\mathcal{M}_{ppr}[L]$ est ordonnable alors L a au plus une branche lourde.

Question 4.6. Soit A un automate émondé qui a au moins deux chemins stricts (distincts) vers des cycles (potentiellement identiques) et soient u et u' des mots étiquetant deux tels chemins.

a. Montrer que u n'est pas un préfixe de u' et que u' n'est pas un préfixe de u.

b. Montrer que $L(A)$ a au moins deux branches lourdes.

Question 4.7. Soit A un automate émondé qui a un seul chemin strict vers un cycle. Montrer que si A n'est pas pseudo-acyclique alors $L(A)$ a au moins deux branches lourdes. En déduire que pour un langage régulier L infini, $\mathcal{M}_{ppr}[L]$ est ordonnable si et seulement si n'importe quel automate émondé A acceptant L est un automate poêle.

Fin du sujet.