DS1 (3 heures)

Lisez tout le texte avant de commencer. La plus grande importance sera attachée à la clarté, à la précision et à la concision de la rédaction. Toute réponse non justifiée ne sera pas prise en compte. Si vous repérez ce qui vous semble être une erreur d'énoncé, signalez la sur votre copie et poursuivez votre composition en expliquant les raisons de vos éventuelles initiatives. L'usage de tout dispositif électronique est interdit.

Le sujet comporte quatre exercices. Vous devez traiter les exercices 1 et 2 et, au choix, soit l'exercice 3, soit l'exercice 4. À titre indicatif, une estimation des durées de traitement et des niveaux de difficultés de chaque exercice est donnée ci-dessous.

- *Exercice 1 : 30 min (*)*
- *Exercice 2 : 30 min (*)*
- *Exercice 3 : 2 h (**)*
- *Exercice 4 : 2 h (***)*

Dans tout ce sujet, OCaml est le seul langage de programmation autorisé. Seules les fonctions incluses dans la bibliothèque standard du langage sont autorisées.

Exercice 1

On considère un ensemble fini de n *variables propositionnelles* $\mathcal{V} = \{p_1, \dots, p_n\}$ et l'ensemble $\mathcal{F}_\mathcal{V}$ des *formules* construites à partir des éléments de $\mathcal{V}$, des connecteurs usuels de conjonction $\wedge$, de disjonction $\vee$ et de négation $\neg$. La formule sans variable propositionnelle, appelée *formule vide* et notée $\top$, est aussi un élément de $\mathcal{F}_\mathcal{V}$. Toutes les formules considérées dans cet exercice sont des formules de $\mathcal{F}_\mathcal{V}$. Pour toute formule ϕ, $\mathcal{V}_\phi$ désigne l'ensemble des variables propositionnelles qui apparaissent dans ϕ.

Un *littéral* est une variable propositionnelle ou bien la négation d'une variable propositionnelle. Le littéral est dit *positif* dans le premier cas, *négatif* dans le second cas. Une *clause* est une formule de la forme $l_1 \vee \cdots \vee l_q$, où $q \geqslant 1$ et $l_1, \dots, l_q$ sont des littéraux deux à deux distincts. Une formule est sous *forme normale conjonctive* si elle s'écrit $C_1 \wedge \cdots \wedge C_m$, où $m \geqslant 0$ et C_1, ..., C_m sont des clauses. Si $m = 0$, on obtient $\top$. Une *formule de Horn* est une formule sous forme normale conjonctive telle que chacune de ses clauses comporte au plus un littéral positif. Une formule est *satisfiable* s'il existe une valuation à valeur dans *{vrai, faux}* de ses variables propositionnelles qui rende la formule vraie. La formule $\top$ est considérée comme satisfiable.

Question 1. Indiquer si les formules suivantes sont satifiables ou non. Dans le cas positif, donner un exemple de valuation des variables propositionnelles qui rende la formule vraie.

□ **1.1.** $(\neg p1 \vee p2) \wedge (p1 \vee \neg p2 \vee \neg p3)$

□ **1.2.** $(p2) \wedge (\neg p1 \vee \neg p3) \wedge (\neg p2) \wedge (p1 \vee \neg p3 \vee \neg p4)$

□ **1.3.** $(p2) \wedge (\neg p1 \vee \neg p2) \wedge (p1 \vee \neg p2) \wedge (p1 \vee \neg p2 \vee \neg p3)$

Question 2. Soit H une formule sous forme normale conjonctive telle que chacune de ses clauses contienne au moins un littéral négatif. Montrer que H est satisfiable en exhibant une valuation de $\mathcal{V}$.

Question 3. Soit H une formule de Horn telle qu'une de ses clauses soit restreinte à un littéral positif p_k, $k \in \{1, \dots, n\}$, et qu'aucune autre de ses clauses ne soit restreinte à $\neg p_k$. À partir de H, montrer que l'on peut construire une formule de Horn H' telle que $\mathcal{V}_{H'} \subset \mathcal{V}_H \setminus \{pk\}$ et que H soit satisfiable si et seulement si H' est satisfiable.

Question 4. Déduire des deux questions précédentes un algorithme qui détermine si une formule de Horn H est satisfiable. Dans le pire des cas, sa complexité doit être majorée par un polynôme en n et m, où n et m désignent respectivement le nombre de variables propositionnelles et le nombre de clauses de H. Vous justifierez ce résultat. Cet algorithme sera explicité sans utiliser de langage de programmation.

Question 5. Appliquer cet algorithme à l'exemple de la question 1.3.

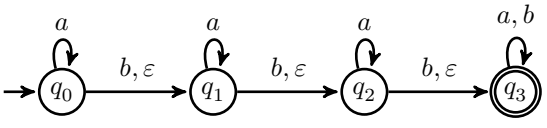
Exercice 2

Question 1. Dans une version simplifiée, le *lemme d'Arden* établit le résultat suivant : *si K et L sont deux langages définis sur un alphabet Σ et si $\varepsilon \notin K$ alors il existe un unique langage X solution de l'équation $X = KX + L$ qui s'écrit sous la forme $X = K^*L$.*

□ **1.1.** À l'aide d'un automate fini et du lemme d'Arden, montrer que $(a|b)^* = (a^*b)^*a^*$ et que $(a|b)^* = (b^*a)^*b^*$.

□ **1.2.** En déduire que $a^*(a|b)^* = (a|b)^*$ et que $b^*(a|b)^* = (a|b)^*$.

Question 2. Dans la suite de l'exercice, Σ désigne l'alphabet $\{a, b\}$. On considère l'automate fini avec transitions spontanées reprèsenté par le graphe suivant.



□ **2.1.** Construire un automate fini non déterministe sans transitions spontanées équivalent.

□ **2.2.** Montrer que le langage reconnu par cet automate peut être dénoté par l'expression régulière $a^*(a|b)^*$.

Exercice 3

Soit Σ un alphabet et Σ^* l'ensemble des mots construits sur Σ. Si $w \in \Sigma^*$, sa *longueur* est notée $|w|$. Si $|w| = n$ est non nul, il existe n caractères $x_0, x_1, \dots, x_{n-1}$ de Σ tels que $w = x_0x_1 \dots x_{n-1}$. Pour tout couple d'entiers (i, j) vérifiant $0 \leqslant i \leqslant j < |w|$, le mot $x_i \dots x_j$ est noté $w[i, j]$. Par convention, si $j < i$, $w[i, j]$ est le *mot vide* ε.
On dit que w est un *carré* s'il existe un mot u de Σ^* tel que $w = uu$. Un mot v de Σ^* est un *facteur* de w s'il existe deux mots r et s de Σ^*, éventuellement vides, tels que $w = rvs$. Le mot w contient une *répétition* s'il contient un facteur carré différent de ε.

Dans la suite, un mot sera représenté en OCaml par la liste de ses caractères. Par exemple, le mot $baba$ est représenté par la liste `[b;a;b;a]` et le mot vide est représenté par la liste `[]`.

Question 1. Écrire une fonction récursive `longueur : 'a list -> int` qui renvoie la longueur de la liste argument.

Question 2. Écrire une fonction `sous_liste : 'a list -> int -> int -> 'a list` telle que `sous_liste lst k len` renvoie la sous-liste de `lst` de longueur `len` qui commence à l'élément de rang `k`. Par convention, le premier élément d'une liste est de rang `0`.

Un algorithme naïf

Question 3. Préciser si les mots suivants contiennent ou non une répétition.

□ **3.1.** $aabpa$ □ **3.2.** $abpdaeq$ □ **3.3.** $ababa$ □ **3.4.** $apba$

Question 4. Soit w un mot contenant au plus deux caractères différentes. Montrer que si $|w| \geqslant 4$ alors w contient au moins une répétition.

Question 5.
□ **5.1.** Écrire une fonction `estCarre : 'a list -> bool` prenant en argument une liste `w` et renvoyant `true` si `w` est un carré, `false` sinon.
□ **5.2.** Déterminer sa complexité en nombre de comparaisons de caractères.

Question 6. Écrire une fonction `contientRepetitionAux : 'a list->int -> bool` prenant en argument une liste `w`, un entier `m` et renvoyant `true` si `w` contient une répétition de la forme `x x` avec `x` de longueur `m`, `false` sinon.

Question 7. Montrer que toute répétition d'un mot w de longueur n est de la forme xx avec $|x| \leqslant \frac{n}{2}$.

Question 8.
□ **8.1.** En déduire une fonction `contientRepetition : 'a list -> bool` prenant en argument une liste `w` renvoyant `true` si `w` contient une répétition, `false` sinon.
□ **8.2.** Déterminer sa complexité en nombre de comparaisons de caractères.

Algorithme de Main-Lorentz

L'algorithme de Main-Lorentz détecte de manière plus efficace des répétitions d'un mot w. Il comporte essentiellement deux parties :

- la première consiste à voir si étant donné deux mots u et v, le mot uv contient un carré non nul issu de la concaténation ;
- la deuxième s'appuie sur le principe de *diviser pour régner*.

Remarquons qu'un mot uv contient une répétition si et seulement si u ou v contiennent une répétition ou uv contient des répétitions provenant de la concaténation. Pour déterminer si un mot uv contient de nouvelles répétitions, on commence par effectuer des prétraitements consistant à calculer des tables de valeurs de u et de v qui sont généralement appelées *tables de préfixes* (ou suffixes). Avant de présenter des algorithmes permettant de générer ces tables, on commence par justifier leur application dans la détection de répétitions.

Soient u et v deux mots de Σ^*. On dit que uv contient un *carré centré* sur u (respectivement sur v) s'il existe un mot w non vide et des mots u', v'', w', w'' tels que $u = u'ww'$, $v = w''v''$, $w = w'w''$ (respectivement $u = u'w'$, $v = w''wv''$, $w = w'w''$).

Soient u et v deux mots de Σ^*. Le *plus long préfixe commun* (respectivement *plus long suffixe commun*) de u et v est le plus long mot w tel qu'il existe deux mots r et s tels que $u = wr$ et $v = ws$ (respectivement $u = rw$ et $v = sw$). On le note $\mathrm{lcp}(u, v)$ (respectivement $\mathrm{lcs}(u, v)$).

Question 9. Dans cette question, $\Sigma = \{a, b\}$. Soient $u = abababaa$ et $v = ababaaa$. Déterminer le plus grand préfixe commun de u et v.

Question 10. Soient u et v deux mots de Σ^*. Montrer que uv contient un carré centré sur u si et seulement s'il existe $i \in \{0, \cdots, |u| - 1\}$ tel que $|\,\mathrm{lcs}(u[0, i-1], u)| + |\,\mathrm{lcp}(u[i, |u|-1], v)| \geqslant |u| - i$.

De la même façon, on montre que uv contient un carré centré sur v si et seulement s'il existe $j \in \{1, \cdots, |v|-1\}$ tel que $|\operatorname{lcs}(v[0, j-1], u)| + |\operatorname{lcp}(v, v[j, |v|-1])| \geqslant |v| - j$.

Ainsi, pour déterminer l'existence d'un carré centré sur u ou v, on peut utiliser les valeurs :

$$|\operatorname{lcs}(u[0, i-1], u)|, |\operatorname{lcp}(u[i, |u|-1], v)|, |\operatorname{lcs}(v[0, j-1], u)|, |\operatorname{lcp}(v, v[j, |v|-1])|$$

Dans la suite, étant donné deux mots u et v, on note pref_u, $\text{pref}_{u,v}$, suff_u et $\text{suff}_{u,v}$ les tableaux vérifiant :

$$\forall i \in \{0, \cdots, |u|-1\} \quad \begin{cases} \text{pref}_u[i] & = |\operatorname{lcp}(u[i, |u|-1], u)| \\ \text{pref}_{u,v}[i] & = |\operatorname{cp}(u[i, |u|-1], v)| \\ \text{suff}_u[i] & = |\operatorname{lcs}(u[0, i], u)| \\ \text{suff}_{u,v}[i] & = |\operatorname{cs}(u[0, i], v)| \end{cases}$$

L'algorithme suivant calcule la table pref_u. On admet que sa complexité est en $O(|u|)$ en nombre de comparaisons de caractères.

Algorithme 1 : calcul de la table pref_u

Entrée : une chaîne de caractères u
Sortie : un tableau pref
$i \leftarrow 0$
pref $\leftarrow$ tableau de taille $|u|$ initialisé à 0
pref$[i] \leftarrow |u|$
$g \leftarrow 0$
pour i *allant de* 1 *à* $|u|-1$ **faire**
  si $i < g$ *et* pref$[i-f] < g-i$ **alors**
    pref$[i] \leftarrow$ pref$[i-f]$
  sinon si $i < g$ *et* pref$[i-f] > g-i$ **alors**
    pref$[i] \leftarrow g-i$
  sinon
    $(f, g) \leftarrow (i, \max(g, i))$
    tant que $g < |u|$ *et* $u[g] == u[g-f]$ **faire**
      $g \leftarrow g+1$
    pref$[i] \leftarrow g-f$

En adaptant cet algorithme, il est également possible de calculer la table $\text{pref}_{u,v}$ en $O(|u|)$ comparaisons de caractères.

Question 11. On pose $u = aabbba$ et $v = abbaab$. Déterminer les tableaux pref_u et $\text{pref}_{u,v}$ sans justification.

Question 12. Appliquer cet algorithme au mot $u = aaabaaabaaab$ et compléter le tableau suivant (en le recopiant sur votre copie) de la façon suivante : pour une valeur i donnée, indiquer les valeurs de f, g, pref $[i]$ à l'issue des instructions internes de la boucle. Par exemple, à l'initialisation, $i = 0$, f n'est pas définie, g vaut 0 et pref$[0] = 12$. Pour $i = 1$, à l'issue des instructions internes à la boucle, $f = 1$, $g = 3$, pref$[1] = 2$.

i	f	g	pref$[i]$
0	-	0	12
1	1	3	2
2	·	·	·
⋮	⋮	⋮	⋮
11	4	12	0

Question 13. Déduire de cet algorithme une procédure calculant suff_u.

Dans la suite, on suppose donné l'algorithme tabpref(u, v) qui prend en argument deux chaînes de caractères u et v et qui renvoie la table $\text{suff}_{u,v}$. On admet que sa complexité est $O(|u|)$ en nombre de comparaisons de caractères.

Question 14.
□ **14.1.** Déduire des questions précédentes un algorithme qui, étant donnés deux mots u et v, renvoie `vrai` s'il existe un carré centré sur u et `faux` sinon.
□ **14.2.** Déterminer sa complexité en nombre de comparaisons de caractères.

Question 15.
□ **15.1.** Déduire des questions précédentes un algorithme récursif qui prend en argument une chaîne de caractères et qui renvoie `vrai` si la chaîne contient une répétition et `faux` sinon.
□ **15.2.** Déterminer sa complexité en nombre de comparaisons de caractères.

Exercice 4

Cet exercice traite de l'ordonnabilité dans des espaces métriques. Dans un tel espace, deux éléments successifs sont à distance bornée. On s'intéresse notamment 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*.

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 les propriétés suivantss.

- **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 $(x, y, z) \in X^3, \delta(x, y) \leqslant \delta(x, z) + \delta(z, y)$.

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 $X' \times X'$. $\mathcal{M}[X']$ est encore un espace métrique, appelé *sous-espace* de $\mathcal{M}$.

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

- s ne contient pas de doublons : 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 sont à distance au plus d : pour tout x_i, x_{i+1} de la suite, $\delta(x_i, x_{i+1}) \leqslant d$.

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.

Si $d \in \mathbb{N}^*$, 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

Dans tout le sujet, l'*alphabet* est $\Sigma = \{a, b\}$ ayant pour seules lettres a et b. Un mot w est une suite finie de lettres $w = \alpha_1 \ldots \alpha_n$. 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, désignées par *distance push-pop* et *distance push-pop-droite*, définies ci-après.

La *distance push-pop*, notée 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).*

La distance *push-pop-droite*, notée 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 mots 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 travaille sur les mots, on considère parfois des programmes produisant une suite (potentiellement infinie) de mots de Σ^*, en utilisant les instructions spéciales suivantes : `popL()`, `popR()`, `pushL(`α`)`, `pushR(`α`)`, `output()` avec $\alpha \in \Sigma$.

Le programme dispose, comme état interne, d'une liste L d'éléments de Σ, interprétée comme un mot de Σ^*. La liste L est initialement vide et représente 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 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 $w_1, w_2, \dots$, où w_1 est le premier mot produit par le programme lors de son exécution, w_2 le deuxième et ainsi de suite.
On appelle 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()` et `pushL(`α`)` pour $\alpha \in \Sigma$.

En OCaml, le type `lettre` suivant est défini.

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

Les listes de type `lettre list` représentent les mots de Σ^*. La tête de la liste correspond à la première lettre du mot. Par exemple, la liste `[A;A;B]` représente le mot aab. Les fonctions suivantes sont supposées définies. Elles manipulent toutes une même variable représentant la liste.

```
pushR : lettre -> unit
pushL : lettre -> unit
popR : unit -> unit
popL : unit -> unit
output : unit -> unit
```

Exemple 3. *Le programme suivant est un programme `push-pop-droite` qui produit la suite de mots $w_1, w_2, \dots$ où w_i est $(aa)^{i-1}b$. Ce programme ne se termine jamais.*

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

Exemple 4. *Le programme suivant est un programme `push-pop` qui produit la même suite que le programme `push-pop-droite` précédent.*

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

Préliminaires

Question 1. Soit $\mathcal{M} = (X, \delta)$ un espace métrique tel que X est un ensemble fini. Montrer que $\mathcal{M}$ est ordonnable.

Question 2. Écrire une fonction `delta_ppr : lettre list -> lettre list -> int` prenant en entrée deux listes 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 3. On considère la suite $s := w_1, w_2, \dots$, où w_i est a^{i^2}. Écrire un programme push-pop-droite qui produit la suite s.

Question 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 $\mathcal{M}_{ppr}$, 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 produite 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 5. Écrire un programme push-pop 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 6. Écrire un programme push-pop qui produit un 1-ordre pour $\mathcal{M}_{pp}[L]$, où $L := a^*b^*$.
Indice : on peut visualiser le langage L comme 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 7. Soit $L := b^*|ab^*$.
□ **7.1.** Écrire un programme push-pop qui produit un 1-ordre pour $\mathcal{M}_{pp}[L]$.
□ **7.2.** Prouver que $\mathcal{M}_{ppr}[L]$ n'est pas ordonnable.

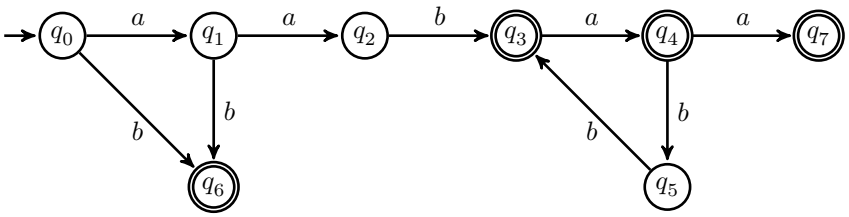
Ordonnabilités des langages réguliers pour la distance push-pop-droite

On souhaite caractériser les langages réguliers ordonnables pour la distance push-pop-droite.

Automates et langages réguliers. Le terme *automate* désigne systématiquement un automate fini déterministe. Formellement, un automate $\mathcal{A} = (Q, q_{\text{init}}, F, t)$ 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* $t : Q \times \Sigma \to Q \cup \{\bot\}$ où $\bot$ signifie que la transition n'est pas définie. Un *chemin* dans $\mathcal{A}$ d'un état $q \in Q$ à un état $q' \in Q$ est une suite finie de la forme $q_0, \alpha_1, q_2, \alpha_2, \dots, \alpha_{n-1}, q_n$ 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 $i \in \{1, \dots, n-1\}$, $q_{i+1} = t(q_i, \alpha_i)$; l'*étiquette* d'un tel chemin est alors le mot $\alpha_1 \dots \alpha_{n-1}$. 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* $\mathcal{A}$, noté $L(\mathcal{A})$, est l'ensemble des mots qui étiquettent 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 $\mathcal{A} = (Q, q_{\text{init}}, F, t)$ 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 $\mathcal{A}$, on peut calculer en temps linéaire un automate $\mathcal{A}'$ émondé tel que $L(\mathcal{A}) = L(\mathcal{A}')$.

Automates poêle. D'après la question 1, si L est fini alors $\mathcal{M}_{ppr}[L]$ est ordonnable. **Dans cette partie, on suppose L infini.** 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 $\mathcal{A} = (Q, q_{\text{init}}, F, t)$ un automate. Un *cycle* dans $\mathcal{A}$ est un chemin de longueur non nulle dans $\mathcal{A}$ 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 $t(q, \alpha) = q$ pour $\alpha \in \Sigma$. On note que si $q_1, \alpha_1, q_2, \alpha_2, \dots, \alpha_{n-1}, q_1$ est un cycle, alors $q_2, \alpha_2, \dots, \alpha_{n-1}, q_1, \alpha_1, q_2$ et plus généralement $q_i, \alpha_i, \dots, \alpha_{n-1}, q_1, \alpha_1, \dots, \alpha_{i-1}, q_i$ pour $2 \leqslant i \leqslant n-2$ sont également des cycles : on dit que ce sont les mêmes cycles *à permutation près*.
Un *chemin strict vers un cycle* dans $\mathcal{A}$ 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 $\mathcal{A}$. Formellement, c'est un chemin $q_1, \alpha_1, \dots, \alpha_{n-1}, q_n$ pour $n \in \mathbb{N}$ avec $q_1 = q_{\text{init}}$ tel que q_n fait partie d'un cycle et pour $1 \leqslant i < n$, 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 $\mathcal{A}$ qu'il est *pseudo-acyclique* s'il a au plus un cycle à permutation près. On note qu'un automate tel que $t(q, a) = t(q, b) = q$ n'est jamais pseudo-acyclique, car q, a, q et q, b, q sont deux cycles différents.
Un automate $\mathcal{A}$ 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 $\{q_0, \dots, q_7\}$, 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 chemins stricts distincts vers un cycle et ne serait donc plus un automate poêle.

Question 8. 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). Aucun n'est demandé mais on attend une explication qui puisse être facilement transformée en un code.

Question 9. Lorsque $\mathcal{A}$ 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.
Si $\mathcal{A}$ est un automate poêle, montrer que $L(\mathcal{A})$ 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 10. On suppose que $\mathcal{A}$ est un automate poêle. Écrire un programme push-pop-droite qui calcule un d-ordre pour $\mathcal{M}_{ppr}[L(\mathcal{A})]$, 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 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 $w_1, w_2, \dots$, 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 $B = w_1, w_2, \dots$ dans T_Σ telle que pour tout $i \in \mathbb{N}$, T_{w_i} contient une infinité de mots de L.

Question 11. Montrer que, pour n'importe quel langage L, si L est infini alors L a au moins une branche lourde.

Question 12. Montrer que, pour n'importe quel langage L, si $\mathcal{M}_{ppr}[L]$ est ordonnable alors L a au plus une branche lourde.

Question 13. Soit $\mathcal{A}$ 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.
□ **13.1.** Montrer que u n'est pas un préfixe de u' et que u' n'est pas un préfixe de u.
□ **13.2.** Montrer que $L(\mathcal{A})$ a au moins deux branches lourdes.

Question 14. Soit $\mathcal{A}$ un automate émondé qui a un seul chemin strict vers un cycle. Montrer que si $\mathcal{A}$ n'est pas pseudo-acyclique alors $L(\mathcal{A})$ 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 $\mathcal{A}$ acceptant L est un automate poêle.