# Devoir maison n°3

Corrigé

⋆ ⋆ ⋆

## 0.1 Premier exemple et miroir

**Question 1** Le langage reconnu par $A_1$ est l'ensemble des mots sur $\Sigma = \{0, 1\}$ ayant un 0 en avant-dernière position.

**Question 2** On peut construire $B = (Q \cup \{q_0\}, \{q_0\}, F, T')$ un $\varepsilon$-AFND tel que $q_0$ est un nouvel état initial, et :

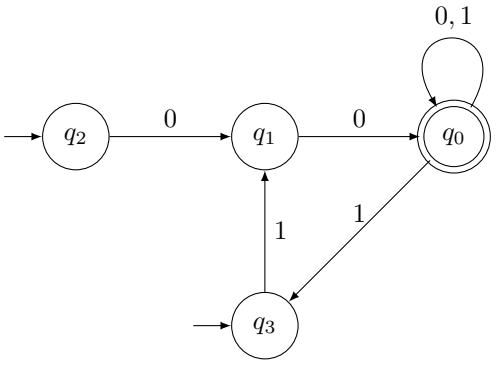
$$T' = T \cup \{(q_0, \varepsilon, q) \mid q \in I\}$$

On rajoute une $\varepsilon$-transition depuis le nouvel état initial vers chaque ancien état initial. Il est clair que $L(A) = L(B)$. De plus, la clôture arrière $B_B$ de $B$ est un AFND sans $\varepsilon$-transition qui reconnaît le même langage et possède les mêmes états initiaux que $B$. Cet automate convient.

**Question 3** On peut poser $L = \{\varepsilon, a\}$ par exemple. Supposons que $L$ est reconnu par un AFND $A = (Q, \{q_0\}, \{q_f\}, T)$.

Sachant que $\varepsilon \in L$, on en déduit que $q_0 = q_f$. De plus, $a$ est reconnu, donc il existe un calcul $q_0 \xrightarrow{a} q_f$ acceptant $a$. Dès lors, on en déduit que pour tout $k \in \mathbb{N}$, $a^k$ est reconnu par $A$, ce qui est absurde.

**Question 4** On peut se contenter d'appliquer la construction décrite juste après.




**Question 5** Montrons que $L(A^T) \subseteq \widetilde{L(A)}$ :

soit $u = a_1 \dots a_n \in L(A^T)$. Alors il existe un calcul dans $A^T$ $q_0 \xrightarrow{a_1} q_1 \dots \xrightarrow{a_{n-1}} q_n$ avec $q_0 \in F$ et $q_n \in I$. On en déduit qu'il existe dans $A$ un calcul $q_n \xrightarrow{a_n} q_{n-1} \dots \xrightarrow{a_1} q_0$. Cela implique que $a_n \dots a_1 = \tilde{u} \in L(A)$, soit $u \in \widetilde{L(A)}$.

Pour l'inclusion réciproque, il suffit de remarquer que $(A^T)^T = A$ et $\widetilde{\widetilde{L(A)}} = L(A)$ et d'appliquer l'inclusion précédente.

**Question 6** La fonction ressemble au calcul d'un graphe transposé : on parcourt chaque liste d'adjacence, et on crée des transitions dans l'autre sens. C'est plus efficace de faire de cette manière que de calculer les listes pour chaque état indépendamment.

```
let transpose aut =
    let n = Array.length aut.delta and
        m = Array.length aut.delta.(0) in
    let deltaT = Array.make_matrix n m [] in
    for q = 0 to n - 1 do
        for a = 0 to m - 1 do
            let traiter q' =
                deltaT.(q').(a) <- q :: deltaT.(q').(a)
            in
            List.iter traiter aut.delta.(q).(a)
        done
    done;
    {init = aut.final;
     final = aut.init;
     delta = deltaT}
```

**Question 7** On remarque que la création de la matrice se fait en temps $\mathcal{O}(|Q| \times |\Sigma|)$. Le remplissage contient une double boucle de taille $\mathcal{O}(|Q| \times |\Sigma|)$ et un parcours de toutes les transitions. Comme la complexité n'est pas demandée en fonction de $|T|$, il faut majorer cette grandeur en fonction de $|Q|$ et $|\Sigma|$. On remarque qu'il y a au plus $|Q|$ transitions étiquetées par une même lettre $a$ partant d'un état $q$ fixé (une vers chaque autre état). On a donc $|T| \leqslant |Q|^2|\Sigma|$, ce qui donne la complexité totale.

## 0.2 Reconnaissance de mot

**Question 8** Il faut parcourir $\Delta(q, a)$ pour chaque élément $q \in X$. Il faut faire attention à ne pas laisser de doublon dans la liste à reconstruire. Pour cela, on peut utiliser un tableau de booléen pour déterminer quels états ont déjà été ajoutés.

Une fois le tableau de booléens construit, il faut le transformer en liste d'éléments.

```
let delta aut ens a =
    let n = Array.length aut.delta in
    let tab = Array.make n false in
    let ajout q = tab.(q) <- true in
    List.iter (fun p -> List.iter ajout aut.delta.(p).(a)) ens;
    let lst = ref [] in
    for q = 0 to n - 1 do
        if tab.(q) then lst := q :: !lst
    done;
    !lst
```

Les appels à `List.iter` parcourent, dans le pire des cas, $|Q|$ listes d'adjacence, donc une complexité en $|Q|^2$ au total. La reconstruction de la liste se fait en temps $\mathcal{O}(|Q|)$.

**Question 9** Cette fonction découle naturellement de la définition de $\Delta^*$.

```
let rec delta_etoile aut ens u = match u with
    | [] -> ens
    | a :: v -> delta_etoile aut (delta aut ens a) v
```

**Question 10** On calcule $\Delta^*(I, u)$, et on vérifie que cet ensemble intersecte $F$. À noter, le test d'intersection pourrait se faire en temps linéaire en $|Q|$ en créant un tableau de booléen, mais le test naïf qui est fait ci-après se fait en $\mathcal{O}(|Q|^2)$, donc est négligeable devant les autres opérations.

```
let reconnu aut u =
    let ens = delta_etoile aut aut.init u in
    List.exists (fun q -> List.mem q aut.final) ens
```

**Question 11** La fonction `delta_etoile` fait un appel à `delta` pour chaque lettre de $u$, soit une complexité en $\mathcal{O}(|Q|^2|u|)$. Comme dit dans la question précédente, c'est cette complexité qui domine dans la fonction `reconnu` (on remarque que $|\Sigma|$ n'intervient pas).

**Question 12** Cela revient à écrire un parcours de graphe dans l'automate. On fait cela via un DFS classique. Une fois le parcours écrit, on lance un appel depuis chaque sommet initial, puis on convertit le tableau des états vus en liste, comme déjà fait précédemment.

```
let accessibles aut =
    let n = Array.length aut.delta and
        m = Array.length aut.delta.(0) in
    let vus = Array.make n false in
    let rec dfs q =
        if not vus.(q) then begin
            vus.(q) <- true;
            for a = 0 to m - 1 do
                List.iter dfs aut.delta.(q).(a)
            done
        end
    in
    List.iter dfs aut.init;
    let lst = ref [] in
    for q = 0 to n - 1 do
        if vus.(q) then lst := q :: !lst
    done;
    !lst
```

**Question 13** Il suffit de vérifier que l'ensemble des états accessibles intersecte l'ensemble des états finaux, comme on l'a fait pour la fonction `reconnu`.

```
let est_vide aut =
    let acc = accessibles aut in
    not (List.exists (fun q -> List.mem q aut.final) acc)
```

# 1 Mots infinis et langages de Büchi

## 1.1 Mots infinis

**Question 14** On pose $u = a_1a_2\ldots a_{m-1}$, pour $m > 1$. On définit $u^\omega = (b_i)_{i\in\mathbb{N}^*}$ tel que pour $i \in \mathbb{N}^*$, $b_i = a_{1+((i-1) \mod (m-1))}$, c'est-à-dire le mot constitué de répétitions du mot $u$. Alors $u^\omega \in \Sigma^\omega$ vérifie bien $u^\omega = uu^\omega$.

Supposons qu'il existe un mot $v = (c_i)_{i\in\mathbb{N}^*} \in \Sigma^\omega$ tel que $v = uv$. Alors $a_1\ldots a_{m-1} = c_1\ldots c_{m-1}$ et $v = (c_i)_{m\leqslant i}$. On en déduit par récurrence que $v = u^\omega$.

**Question 15** Supposons par l'absurde qu'il existe $x, y \in \Sigma^*$ tels que $u = xy^\omega$. Notons $k_0 = |y|_0$ le nombre de 0 dans $y$ et $k_1 = |y|_1$ le nombre de 1 dans $y$. Distinguons :

– si $k_1 = 0$ ou $k_0 = 0$, alors $xy^\omega$ ne contient que des 0 ou que des 1 à partir d'un certain rang, donc $xy^\omega \neq u$ ;

– sinon, notons $p_i$ le préfixe de taille $i$ de $xy^\omega$. Alors $\dfrac{|p_i|_1}{|p_i|_0} \underset{i\longrightarrow\infty}{\longrightarrow} \dfrac{k_1}{k_0}$. C'est absurde car si $p_i$ est un préfixe de $u$, $\dfrac{|p_i|_1}{|p_i|_0} \underset{i\longrightarrow\infty}{\longrightarrow} 0$.

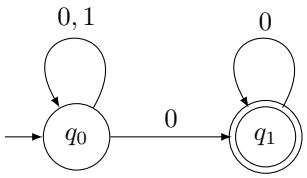
Dans tous les cas, on arrive à une contradiction.

**Question 16** On pose $L = \{a, b\}$. Alors $\{u^\omega \mid u \in L\} = \{a^\omega, b^\omega\}$ est un langage de cardinal 2, mais $L^\omega = \Sigma^\omega$ est un langage de cardinal infini (qui contient tous les mots infinis).

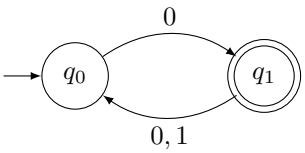
## 1.2 Langage de Büchi

**Question 17**

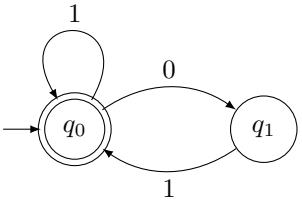
1. On propose, pour $L_1$ :




2. Pour $L_2$ (on fait attention que les indices commencent à 0) :




3. Pour $L_3$ :




**Question 18** Le langage reconnu est l'ensemble des mots ayant soit aucun 0, soit une infinité de 0. La restriction de l'automate à $\{q_0, q_3\}$ n'a pas de transition étiquetée par 0 donc reconnaît la première partie du langage. De plus, on remarque que la restriction à $\{q_1, q_2\}$ est l'automate donné en exemple.

Réciproquement, ces deux types de mots sont bien reconnus par l'automate.

**Question 19** Supposons qu'il existe un automate déterministe $A = (Q, q_0, F, T)$ tel que $\mathcal{B}(A) = L_1$.

Comme $0^\omega \in L_1$, il existe $n_1 \in \mathbb{N}$ tel que $q_1 = \delta^*(q_0, 0^{n_1}) \in F$.

Comme $0^{n_1}10^\omega \in L_1$, il existe $n_2 \in \mathbb{N}$ tel que $q_2 = \delta^*(q_1, 10^{n_2}) \in F$.

On construit de même par récurrence des états $q_i$, $i \geqslant 3$, ... Comme $F$ est fini, il existe $1 \leqslant i < j$ tels que $q_i = q_j$. Dès lors, $\delta^*(q_i, 10^{n_{i+1}}10^{n_{i+2}}1\ldots10^{n_j}) = q_j = q_i$.

On en déduit que $0^{n_1}1\ldots10^{n_i}\,(10^{n_{i+1}}10^{n_{i+2}}1\ldots10^{n_j})^\omega \in \mathcal{B}(A)$, ce qui est absurde car ce mot contient une infinité de 1.

## 1.3 Problème du vide et clôture

**Question 20** Décrire en français un algorithme permettant de déterminer si le langage de Büchi d'un automate est vide ou non. Quelle est sa complexité temporelle ?

**Question 21** On pose $A = (Q_A, I_A, F_A, T_A)$ et $A' = (Q'_A, I'_A, F'_A, T'_A)$. On définit $A_\cup = (Q, I, F, T)$ par :

- $Q = Q_A \cup Q'_A \cup \{q_0\}$ ;
- $I = \{q_0\}$
- $F = F_A \cup F'_A$ ;
- $T = T_A \cup T'_A \cup \{(q_0, a, q') \mid a \in \Sigma \text{ et } \exists q \in I_A \cup I'_A, (q, a, q') \in T_A \cup T'_A\}$

L'idée est de rajouter un état initial et de faire des transitions depuis cet état vers chacun des voisins des états initiaux des deux automates. La lecture d'un mot infini se fera alors dans l'un ou dans l'autre.

**Question 22** On pose $A = (Q_A, I_A, F_A, T_A)$ et $A' = (Q'_A, I'_A, F'_A, T'_A)$. On définit $A_\bullet = (Q, I, F, T)$ par :

- $Q = Q_A \cup Q'_A$ ;
- $I = I_A$ ;
- $F = F'_A$ ;
- $T = T_A \cup T'_A \cup \{(q, a, q'') \mid q \in F_A, a \in \Sigma \text{ et } \exists q' \in I'_A, (q', a, q'') \in T'_A\}$

On rajoute une transition depuis les états finaux du premier automate vers chacun des voisins de l'état initial du deuxième automate pour considérer la concaténation des langages.

**Question 23** On pose $A = (Q, I, F, T)$. Sans perte de généralité, on peut supposer qu'aucune transition ne sort d'un état de $F$ et que $F = \{q_f\}$ (quitte à rajouter une $\varepsilon$-transition vers un nouvel état final et à prendre la clôture avant).

On définit alors $A_\omega = (Q, I, F, T')$ où $T' = T \cup \{(q_f, a, q') \mid a \in \Sigma \text{ et } \exists q \in I_A \cup I'_A, (q, a, q') \in T_A \cup T'_A\}$.

L'idée est qu'on rajoute une transition depuis l'unique état final vers chaque voisin d'un état initial. Cela permet de recommencer la lecture d'un nouveau mot après en avoir terminé un.

⋆ ⋆ ⋆