Devoir maison n°4

Corrigé

⋆ ⋆ ⋆

Question 1 Il est possible que cette condition soit vérifiée, par exemple avec $u = 00$ et $v = 000$. On a alors $k = 2$, et en choisissant $i = 2$ et $i' = 3$, qui sont bien deux occurrences de u dans v, on a bien $i < i'$ et $i \geqslant i' - k + 1 = 3 - 2 + 1 = 2$.

Question 2 Remarquons que la première occurrence de u dans v ne peut être qu'après la position k et que la dernière ne peut être qu'avant la position n. On en déduit que le nombre d'occurrences est majoré par $n - k + 1$. Cette valeur est atteinte pour $u = 0^k$ et $v = 0^n$.

Question 3 On répond dans le cas où l'un des deux mots est vide. Sinon, on compare les premières lettres et on relance éventuellement un appel récursif sur les queues.

```
let rec prefixe u v = match u, v with
    | [], _            -> true
    | _, []            -> false
    | a :: u', b :: v' -> a = b && prefixe u' v'
```

La complexité est traitée à la question 5.

Question 4 On commence par déterminer la longueur du mot u, puis on utilise une fonction auxiliaire qui prend en arguments un entier et un mot et qui teste si u est préfixe de ce mot. Si c'est le cas, cela signifie qu'on a trouvé une nouvelle occurrence de u dans v. L'entier permet de repérer à quelle position se trouve cette occurrence (la première ne pouvant pas se trouver avant k).

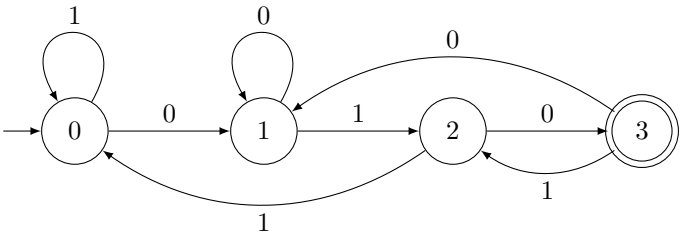
```
let recherche_naive u v =
    let k = List.length u in
    let rec facteur i = function
        | []     -> []
        | a :: w ->
            if prefixe u (a :: w) then
                i :: facteur (i + 1) w
            else
                facteur (i + 1) w in
    facteur k v
```

Un appel à `facteur i x` détermine si u est un facteur de v terminant à l'indice i, en supposant que x est un préfixe de taille $n - i + k$ de v.

Question 5 La fonction `List.length` a une complexité en $\mathcal{O}(k)$ sur un mot de taille k. La fonction `prefixe` a une complexité $\mathcal{O}(\min(k, n))$ pour u et v de tailles k et n. La fonction `recherche_naive` fait au plus n appels récursifs à `prefixe`, donc la complexité totale est en $\mathcal{O}(k + \sum_{i=1}^{n} \min(k, i))$, soit $\mathcal{O}(kn)$.

Question 6 On remarque que par définition, la suite $(\rho^j(q))_{j\in\mathbb{N}}$ est strictement décroissante tant que $\rho^j(q) \neq 0$. L'ensemble $\mathbb{N}$ étant bien fondé, on en déduit que la suite est ultimement constante égale à 0. En particulier, $\rho^k(q) = 0$ pour tout état $q \in Q_{\mathcal{A}}$. Comme pour tout $a \in \Sigma$, $\delta(0, a)$ est bien défini, on en déduit que $\delta(\rho^k(q), a)$ est toujours défini, ce qui montre l'existence de l'entier i demandé.

Question 7 L'automate suivant convient.



L'idée est la suivante : on écrit les transitions déjà existantes, et on rajoute les transitions manquantes en se basant sur le repli :

- pour la lettre 0 depuis l'état 1, on se replie à l'état 0, puis on lit un 0 pour revenir en 1 ;
- pour la lettre 1 depuis l'état 2, on se replie à l'état 0, puis on lit un 1 pour rester en 0 ;
- pour la lettre 0 depuis l'état 3, on se replie à l'état 1, puis à l'état 0, puis on revient en 1 en lisant 0 ;
- pour la lettre 1 depuis l'état 3, on se replie à l'état 1, puis on arrive en 2.

Question 8 Le langage est celui des mots qui finissent par 010, c'est-à-dire $L(\mathcal{A}_1) = \Sigma^*\{010\}$.

Question 9 La difficulté ici est de copier correctement la matrice qui correspond à la fonction de transition. On utilise `Array.copy` sur chaque composante pour éviter les liaisons. On copie les deux autres tableaux avant de créer l'automate à renvoyer.

```
let copie_afdr aut =
    let t = Array.map Array.copy aut.transition and
        f = Array.copy aut.final and
        r = Array.copy aut.repli in
    {final = f; transition = t; repli = r}
```

Question 10 On commence par copier complètement l'automate grâce à la fonction précédente. Ensuite, dans l'ordre croissant des états q, pour chaque lettre a pour laquelle une transition n'est pas définie, on copie la transition $\delta(\rho(q), a)$ (qui a éventuellement été modifié dans un précédent passage dans la boucle). La complexité de la fonction `copie_afdr` est en $\mathcal{O}(k \times m)$ ($k+1$ copies de tableaux de taille m, plus deux copies de tableaux de taille $k+1$). De par la taille des boucle `for` dans la fonction ci-dessous, et parce que les opérations qui sont à l'intérieur sont en temps constant, on a bien à nouveau une complexité $\mathcal{O}(k \times m)$.

```
let enleve_repli aut =
        let k' = Array.length aut.final
        and m = Array.length aut.transition.(0) in
        let afdc = copie_afdr aut in
        let tab = afdc.transition in
        for q = 1 to k' - 1 do
            for a = 0 to m - 1 do
                if tab.(q).(a) = -1 then
                    let q' = aut.repli.(q) in
                    tab.(q).(a) <- tab.(q').(a)
            done
        done;
        afdc
```

Question 11 On se contente de lire les lettres du mot u une à une dans l'AFDC $\mathcal{A}$, et de vérifier pour chaque lettre si l'état dans lequel on se trouve est final ou non.

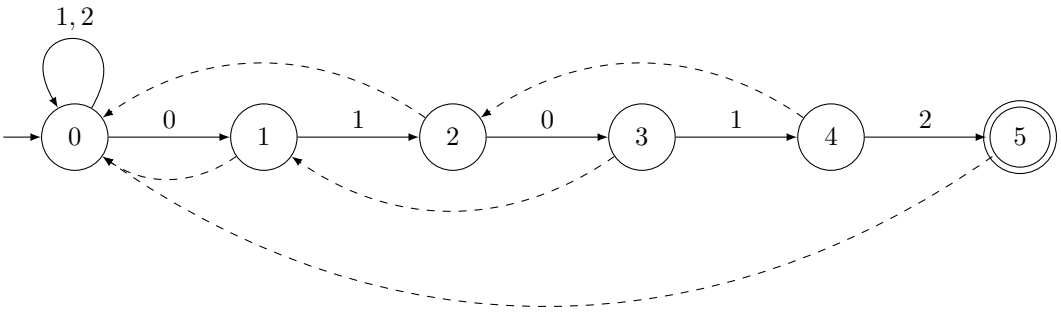
Entrée : AFDC (k, F, δ, ρ), mot $u = u_1 \dots u_n$
Début algorithme
  $q \longleftarrow 0$
  $L \longleftarrow \emptyset$
  Pour $i = 1$ à n **Faire**
    $q \longleftarrow \delta(q, u_i)$
    Si $q \in F$ **Alors**
      $L \longleftarrow L \cup \{i\}$
  Renvoyer L

Question 12 On applique ce qui est décrit précédemment (en récursif plutôt qu'itératif, car les mots ne sont pas indexables).

```
let occurrences aut u =
    let rec lecture i q = function
        | []      -> []
        | a :: u' ->
            let p = aut.transition.(q).(a) in
            if aut.final.(p) then
                i :: lecture (i + 1) p u'
            else
                lecture (i + 1) p u' in
    lecture 1 0 u
```

Ici, un appel `lecture i q u` détermine si la lecture des i premières lettres de u a mené à un état final. On lit uniquement la première lettre, et on lit le reste du mot récursivement.

Question 13 On obtient l'automate suivant :



Question 14 $\mathcal{A}_u^{\text{KMP}}$ reconnait le langage $\Sigma^*\{u\}$.

Question 15 Remarquons au préalable qu'un suffixe non vide de $u_1 \dots u_i$ est un suffixe de $u_1 \dots u_{i-1}$ auquel on a rajouté u_i. Rappelons également que $u_1 \dots u_{\rho(i-1)}$ est le plus long suffixe (de cette forme) de $u_1 \dots u_{i-1}$. Les suffixes stricts de $u_1 \dots u_{i-1}$ de la forme $u_1 \dots u_\ell$ sont donc exactement les $u_1 \dots u_{\rho^j(\rho(i-1))}$, pour $j \in \mathbb{N}$, et en particulier pour $j \in [\![0, j_i]\!]$ (notons que si $\rho^j(\rho(i-1)) = 0$, alors le suffixe considéré est le mot vide ε). Ainsi, comme la suite des $(\rho^j(\rho(i-1)))_j$ est strictement décroissante puis constante égale à 0, on en déduit que le plus long suffixe strict de $u_1 \dots u_i$ vérifie :

- soit il est vide, auquel cas $\delta(\rho^j(\rho(i-1)), u_i)$ n'est défini que pour $\rho^j(\rho(i-1)) = 0$ et vaut 0. Dans ce cas, on a bien $j = j_i$ et $\rho(i) = 0 = \delta(\rho^p(\rho(i-1)), u_i)$;
- soit il est non vide, auquel cas il est de la forme $u_1 \dots u_{\rho^j(\rho(i-1))}$, avec $u_{\rho^j(\rho(i-1))+1} = u_i$, on en déduit $\rho(i) = \rho^j(\rho(i-1)) + 1 = \delta(\rho^j(\rho(i-1)), u_i)$. Dans ce cas, pour tout $p < j$, $u_{\rho^p(\rho(i-1))+1} \neq u_i$, donc $\delta(\rho^p(\rho(i-1)), u_i)$ n'est pas défini, et on a bien à nouveau $j = j_i$.

Question 16 On utilise la formule précédente pour remplir la fonction de repli. Dans la fonction ci-dessus, on commence par initialiser toutes les transitions depuis 0 vers 0 et les autres étant des blocages. La fonction

auxiliaire `modif_transitions` prend en arguments un indice i (correspondant à l'indice de la lettre en cours de lecture) et le mot $u_i \dots u_k$. On traite à part le cas $i = 1$ (car la formule précédente n'est pas vérifiée), en posant $\delta(0, u_1) = 1$ et $\rho(1) = 0$, et si $i \geqslant 2$, on pose $\delta(i-1, u_i) = i$ et on cherche la valeur de j_i (à l'aide d'une boucle `while`) pour remplir la valeur de $\rho(i)$.

```
let kmp u m =
    let k = List.length u in
    let f = Array.make (k + 1) false in
    let t = Array.make_matrix (k + 1) m (-1) in
    let r = Array.make (k + 1) 0 in
    f.(k) <- true;
    for a = 0 to m - 1 do
        t.(0).(a) <- 0;
    done;
    let rec modif_transitions i = function
        | []                  -> ()
        | a :: u' when i = 1 -> t.(0).(a) <- 1; r.(1) <- 0; aux 2 u'
        | a :: u'            ->
            t.(i - 1).(a) <- i;
            let q = ref r.(i - 1) in
            while t.(!q).(a) = -1 do
                q := r.(!q)
            done;
            r.(i) <- t.(!q).(a);
            modif_transitions (i + 1) u' in
    modif_transitions 1 u;
    {final = f; transition = t; repli = r}
```

Question 17 On sait que $\rho(i) = \delta(\rho^{j_i}(\rho(i-1)), u_i)$. Or, $\delta(\rho^{j_i}(\rho(i-1)), u_i)$ vaut 0 ou $\rho^{j_i}(\rho(i-1)) + 1$, donc $\rho(i) \leqslant \rho^{j_i}(\rho(i-1)) + 1$. Enfin, comme $\rho(x) < x$ si $x \neq 0$, par une récurrence rapide, on montre que $\rho^\ell(x) \leqslant x - \ell$ tant que $\rho^{\ell-1}(x) \neq 0$.

Finalement, on a $\rho(i) \leqslant \rho^{j_i}(\rho(i-1)) + 1 \leqslant \rho(i-1) + 1 - j_i$.

On en déduit que $j_i \leqslant \rho(i-1) - \rho(i) + 1 \leqslant 1$ (car $\rho(i) \geqslant \rho(i-1)$, soit finalement $\sum_{i=1}^{k} j_i \leqslant k = \mathcal{O}(k)$.

Question 18 La création des tableaux se fait en $\mathcal{O}(km)$. La boucle `for` se fait en $\mathcal{O}(m)$. La fonction auxiliaire fait un appel récursif pour i de 1 à k, et pour chaque appel récursif, on a une boucle de taille j_i et des opérations en temps constant. On en déduit que l'appel `modif_transitions 1 u` s'exécute en temps $\mathcal{O}(\sum_{i=1}^{k} j_i) = \mathcal{O}(k)$. La complexité totale est donc en $\mathcal{O}(km)$.

Question 19 On commence par calculer la taille m de l'alphabet, en écrivant une fonction qui cherche l'élément maximal dans une liste. Ensuite, on construit l'automate KMP associé à u et on utilise la fonction `occurrences` écrite précédemment en ayant au préalable complété l'automate.

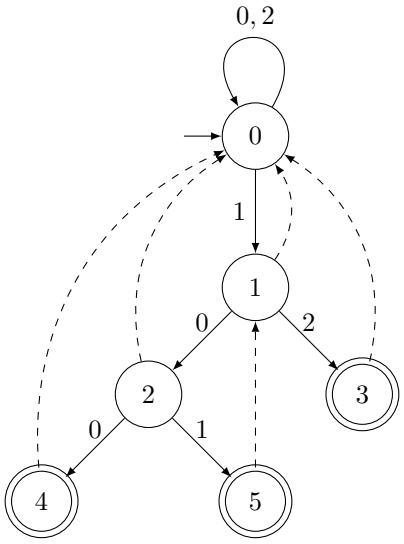
```
let rec maximum = function
    | []     -> -1
    | x :: q -> max x (maximum q)

let recherche_kmp u v =
    let m = max (maximum u) (maximum v) + 1 in
    let aut = kmp u m in
    occurrences (enleve_repli aut) v
```

La fonction `maximum` a une complexité linéaire en la taille de la liste. On en déduit que la recherche de m se fait en $\mathcal{O}(k+n)$. La construction de l'automate KMP se fait en $\mathcal{O}(km)$. Sa complétion se fait en $\mathcal{O}(km)$ également, et le calcul des occurrences se fait ensuite en $\mathcal{O}(n)$. On en déduit une complexité totale en $\mathcal{O}(km+n)$,

là où la recherche naïve se faisait en $\mathcal{O}(kn)$. Comme la taille de l'alphabet est généralement petite devant la taille des mots considérés, on en déduit que cette complexité est bien plus intéressante.

Question 20 Ce langage reconnait les mots qui terminent par 00, 01 ou 10.

Question 21 L'automate suivant convient. On s'inspire de la construction des automates KMP, en envisageant plusieurs branches.



Question 22 Pour chaque mot u du dictionnaire D, on détermine la liste des occurrences de u dans v. Une fois calculée, on la concatène avec la liste des occurrences des autres mots du dictionnaire.

```
let rec recherche_dico_kmp dico v = match dico with
    | [] -> []
    | u :: d -> recherche_kmp u v @ recherche_dico_kmp d v
```

La concaténation ayant une complexité linéaire en la taille de la première liste, qui ici sera de taille au plus n, on en déduit les détails de la complexité :

- $|U|$ concaténations, soit une complexité en $\mathcal{O}(|D|n)$;
- $|U|$ appels à `recherche_kmp`, soit une complexité en $\mathcal{O}(|D|(n+km))$.

La complexité totale est donc en $\mathcal{O}(|D|(n+km))$.

Question 23 L'idée est de construire un automate de dictionnaire en s'inspirant des automates précédents selon le principe suivant :

- on crée un état pour chaque préfixe d'un mot de D ;
- chaque mot de D est un état final ;
- pour chaque préfixe x d'un mot de D, pour chaque lettre $a \in \Sigma$, on crée la transition $\delta(x,a) = xa$ si et seulement si xa est un préfixe d'un mot de D ;
- pour chaque lettre $a \in \Sigma$, on crée la transition $\delta(\varepsilon, a) = \varepsilon$ si et seulement si a n'est pas un préfixe d'un mot de D ;
- pour chaque préfixe x d'un mot de D, on pose $\rho(x)$ est le plus long suffixe strict de x qui est aussi un préfixe d'un mot de D.

Une fois cet automate construit, on peut le compléter puis lui appliquer la fonction `occurrences` pour calculer toutes les occurrences d'un mot de D dans v.

Les difficultés viennent bien sûr de la création de l'automate. En particulier, il faudrait un moyen d'associer un entier à chaque préfixe, ainsi qu'une technique pour calculer efficacement les replis. Remarquons aussi que le nombre d'états est au plus $k|D|+1$. On peut espérer avoir ainsi une complexité totale en $\mathcal{O}(km|D|+n)$ qui serait meilleur que l'algorithme précédent.

Pour plus de détails, se renseigner sur l'algorithme d'Aho-Corasick, qui est à la base de la recherche `fgrep` dans les systèmes UNIX.

⋆ ⋆ ⋆