TD3 - Automates finis (éléments de réponses)

Exercice 1

Fait en td.

Exercice 2

Fait en td.

Exercice 3

Fait en td.

Exercice 4

Fait en td.

Exercice 5

Par rapport au sujet initial, les fonctions ont été renommées, de manière assez évidente. Cela prépare le futur de ce sujet...

Liste de succ

Question 1.

```
let rec add q lst = match lst with
  | [] -> [q]
  | x :: t when x = q -> lst
  | x :: t when x < q -> x :: (add q t)
  | _ -> q :: lst

let rec merge lst1 lst2 = match lst1, lst2 with
  | [], [] -> []
  | _ , [] -> lst1
  | [], _ -> lst2
  | x1 :: t1, x2 :: t2 when x1 = x2 -> x1 :: (merge t1 t2)
  | x1 :: t1, x2 :: t2 -> if x1 < x2
                          then x1 :: (merge t1 lst2)
                          else x2 :: (merge lst1 t2)

let rec eql lst1 lst2 = match lst1, lst2 with
  | [], [] -> true
  | x1 :: t1, x2 :: t2 -> (x1 = x2) && (eql t1 t2)
  | _, _ -> false

let rec intersect lst1 lst2 = match lst1, lst2 with
  | _, [] -> []
  | [], _ -> []
  | x1 :: q1, x2 :: q2 when x1 = x2 -> x1 :: (intersect q1 q2)
  | x1 :: q1, x2 :: q2 -> if x1 < x2
                          then intersect q1 lst2
                          else intersect lst1 q2
```

□ **1.1.** Complexités.

- `ajoute` : $O(|\texttt{lst}|)$
- `fusion` : $O(|\texttt{lst1}| + |\texttt{lst2}|)$
- `egalite` : $O(|\texttt{lst1}| + |\texttt{lst2}|)$
- `intersecte` : $O(|\texttt{lst1}| \times |\texttt{lst2}|)$

Question 2.

```
let succ q af =
  let rec aux lst_trans = match lst_trans with
    | [] -> []
    | (q1, _, q2) :: t when q1 = q -> add q2 (aux t)
    | _ :: t -> aux t
  in aux af.trans
```

```
let pred q af =
  let rec aux lst_trans = match lst_trans with
    | [] -> []
    | (q1, _, q2) :: t when q2 = q -> add q1 (aux t)
    | _ :: t -> aux t
  in aux af.trans
```

Complexité en $O(n \times p)$.

Question 3.

```
let rec extend lst af gamma = match lst with
  | [] -> []
  | q :: t -> let lst1 = gamma q af
              and ext1 = extend t af gamma
              in add q (merge lst1 ext1)
```

La fonction γ étant soit la fonction succ, soit la fonction pred, elle renvoie une liste d'états de Q. Considérons une suite d'ensembles d'états $E_0 \subset E_1 \subset E_2 \subset \cdots \subset E_k \subset \ldots$ susceptibles d'être renvoyés par la fonction etend. Pour tout entier naturel k, $E_k \subset Q$. En outre, cette suie est croissante pour l'inclusion. L'ensemble Q étant fini, cette suite est stationnaire au bout d'au plus $n = |Q|$ étapes. Par conséquent, $E_{n+1} = E_n$ et la fonction s'arrête.
La complexité de etend est en $O(n^2 p)$.

Question 4.

```
let access af gamma lst_q =
  let rec aux lst =
    let ext_lst = extend lst af gamma
    in if eql lst ext_lst then lst else aux ext_lst
  in aux lst_q
```

Question 5.

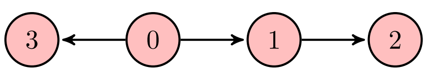
```
tch trans with
  | [] -> []
  | (q1, x, q2) :: t when not (List.mem q1 lst_q) || not (List.mem q2 lst_q) ->
     rm_trans lst_q t
  | (q1, x, q2) :: t -> (q1, x, q
```

Question 6. Le bilan de cette partie est qu'on dispose d'un moyen d'émonder un automate : on construit la liste de tous ses états utiles (les différentes fonctions construites ont pour objectif d'arriver à atteindre cet objectif) et on reconstruit un automate en ne conservant que ces états utiles et les transitions entre eux, quand elles existent.

```
let trim af =
  let acc = access af succ af.init in
  let coacc = access af pred af.finals in
  let lst_q = intersect acc coacc in
  let init = intersect af.init lst_q in
  let finals = intersect af.finals lst_q in
  let new_trans = rm_trans lst_q af.trans in
    {alphabet = af.alphabet; etats = List.length lst_q;
     init = init; finals = finals; trans = new_trans}
```

Algorithme de Warshall

L'idée de l'algorithme de Warshall, également appelé algorithme de Roy-Warshall, est de construire itérativement une matrice dont les éléments précisent l'existence ou non d'un chemin entre deux sommets d'un graphe. Une variante de cet algorithme est celui de Floyd-Warshall qui calcule une plus courte distance entre deux sommets s'il existe un chemin les reliant. En revanche, aucun des deux algorithmes ne permet, en l'état, de trouver les chemins.
Pour cerner l'algorithme de Warshall, une illustration peut aider. Considérons le graphe ci-dessou et sa matrice d'adjacence C, les 1 indiquant la présence effective d'un chemin, les 0 son absence.



$$C = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

L'algorithme de Warshall construit successivement les matrices suivantes.

$$C_0 = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad C_1 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad C_2 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad C_3 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Il s'arrête avec le calcul de C_3 qui informe sur l'existence de chemins entre les sommets suivants.

$$0 \to 1 \qquad 0 \to 2 \qquad 0 \to 3 \qquad 1 \to 2$$

Question 7.

```
let mat_adj af =
  let n = af.etats in
  let a = Array.make_matrix n n false in
  let rec aux t = match t with
    | [] -> ()
    | (i, e, j) :: q -> a.(i).(j) <- true; aux q
  in for i = 0 to n - 1 do a.(i).(i) <- true done;
  aux af.trans;
  a
```

Question 8. S'il existe un chemin pour aller de i à j en passant par les sommets de $[\![0, k]\!]$ ($C_k[i, j] =$ `true`), c'est que soit il en existe déjà un en ne passant que par les sommets de $[\![0, k-1]\!]$, soit il en existe un en passant par le sommet k. La réciproque est immédiate.

Question 9.

□ **9.1.** Si $C_k[i, k+1]$ est vrai, on a également $C_{k+1}[i, k+1]$ qui est vrai. Réciproquement, si $C_{k+1}[i, k+1]$ est vrai, on peut trouver un chemin $q_i \xrightarrow{a_1} \dots q_{k+1}$ en ne passant que par des états $0, \dots, k+1$. Il suffit de considérer le premier état égal à $(k+1)$ dans ce chemin. Pour atteindre cet état, on ne passe que par des états intermédiaires numérotés $0, \dots, k$, autrement dit $C_k[i, k+1]$ est vrai.

□ **9.2.** La démonstration est similaire en considérant le premier état $(k+1)$ dans un chemin.

Question 10.

□ **10.1.** Fonction `matrice_accessibilite`.

```
let matrice_accessibilite af =
  let c = mat_adj af in
  let n = Array.length c in
  for k = 0 to n - 2 do
    for i = 0 to n - 1 do
      for j = 0 to n - 1 do
        if (not c.(i).(j)) && c.(i).(k) && c.(k).(j)
        then c.(i).(j) <- true
      done;
    done;
  done;
  c
```

□ **10.2.** D'après la question précédente, il suffit de mettre à vrai $C_k[i, j]$ lorsque $C_{k-1}[i, j]$ est faux et $C_{k-1}[i, k]$ et $C_{k-1}[k, j]$ sont vrais. L'algorithme est en $O(n^3)$ puisqu'il faut calculer n matrices et chaque calcul de matrices nécessite $2n^2$ opérations booléennes.

Question 11.

```
let deletat i c =
  let rec aux j = match j with
    | 0 when c.(i).(0) -> [0]
    | 0 -> []
    | _ when c.(i).(j) -> add j (aux (j - 1))
    | _ -> aux (j - 1) in
  let n = Array.length c
  in aux (n - 1)

let rec accessibles l c = match l with
  | [] -> []
  | i :: q -> merge (deletat i c) (accessibles q c)
```

```
let etats_acc af =
  let c = matrice_accessibilite af
  and init = af.init
  in accessibles init c
```