TP ENS

Automates, bisimilarité, équivalence

Hasley William

0.1 Automates

Enjeu du sujet :
Ce sujet s'intéresse aux automates, et plus particulièrement à la notion de Bisimilarité entre deux automates, symbolisant en somme l'équivalence dans la lecture de ces deux automates.

Comme toujours, ce sujet comportera tout du long un aspect aléatoire via un générateur pseudo-aléatoire. Cet aspect aléatoire servira à générer une famille d'automates.

Question 1. (calcul de la suite u)

```
(* Definition de constantes globales donnees par le sujet *)
let max_u = 99999
let u_tab = Array.make max_u (-1)
let u_0 = 1
let u_def_index = ref 0
let a = 1103515245
let c = 12345
let m = 0b1000000000000000

(* Gestion du caractere Pseudo-Aleatoire via le tableau u_tab, Interface avec get_u *)

let get_u i =
  if (i >= max_u) then failwith "get_u -> Outside of Range!\n"
  else if (i <= !u_def_index) then u_tab.(i) else
  begin
    for k = !u_def_index + 1 to i do
      u_tab.(k) <- (a * u_tab.(k-1) + c) mod m
    done;
    u_def_index := i;
    u_tab.(i)
  end


(* Question 1 *)

let q_1 i =
    Printf.printf "   * Question 1 : u_%d = %d \n" i (get_u i)
```

Question 2. (Génération de la famille d'automates aléatoire)

Le sujet souhaite s'intéresser aux automates définis par une liste de triplets symbolisant les transitions. Nous choisissons d'adapter cette définition en représentant nos automates sous la forme d'un quintuplet $(Q, \Sigma, \delta, I, F)$, où :

- Q : int représente le nombre d'états de l'automate ($Q \sim [\![0, |Q|-1]\!]$)
- Σ : int représente la taille de l'alphabet de l'automate.
- δ : transition_t array est un tableau contenant pour chaque état la liste des transitions partant de cet état (une transition est un couple lettre × etat).
- I : etat_t list est une liste des états initiaux.
- F : etat_t list est une liste des états acceptants.

Ci-dessous la définition des types mis en jeu, et de quelques primitives de base comme l'identité ou les projections usuelles.

```
(* Definition des types Automates, Alphabet, ...*)
(* Il n'est en soi pas necessaire de developper ainsi, mais ceci rend plus clair le debug / verbose*)

type alphabet_t = int (* Assimile a [|0, n|]*)
type etat_t = int (* Assimilea [|0, n|]*)
type transition_t = alphabet_t * etat_t
type list_transition_t = transition_t list array (* Chaque etat possede sa liste de transitions, le sujet
    souhaite aborder la question des transition sous la forme de liste de triplets plutot qu'en fonctionnel *)
```

```
type etat_init_t = etat_t list
type etat_fin_t = etat_t list
type automate_t = etat_t * alphabet_t * list_transition_t * etat_init_t * etat_fin_t


let transition_identity (t : transition_t) : transition_t = t

let transition_get_letter (t : transition_t) : alphabet_t = match t with
  |(l, n) -> l

let transition_get_next (t : transition_t): etat_t = match t with
  |(l, n) -> n


let aut_identity (a : automate_t) : automate_t = a

let aut_get_states (a : automate_t) : etat_t = match a with
  |(q, sigma, delta, i, f) -> q

let aut_get_alphabet (a : automate_t) : alphabet_t = match a with
  |(q, sigma, delta, i, f) -> sigma

let aut_get_trans_arr (a : automate_t) : list_transition_t = match a with
  |(q, sigma, delta, i, f) -> delta

let aut_get_init_state (a : automate_t) : etat_init_t = match a with
  |(q, sigma, delta, i, f) -> i

let aut_get_fin_state (a : automate_t) : etat_fin_t = match a with
  |(q, sigma, delta, i, f) -> f


let aut_affiche (a : automate_t) : unit = match a with
  |(q, sigma, delta, i, f) ->
    Printf.printf "Automate : q = %d, Sigma = %d : \n" q sigma;

    Printf.printf "Etats Initiaux : "; List.iter (fun i -> Printf.printf "%d " i) i;
    Printf.printf "\nEtats Terminaux : "; List.iter (fun i -> Printf.printf "%d " i) f; print_newline ();

    Array.iteri (fun i li -> List.iter (fun (l, n) -> Printf.printf "%d ->_%d %d" i l n) li; print_newline ())
        delta
```

Ceci donne ainsi pour la génération des automates aléatoires :

```
let x (i : int) (j: int) (s: int) : int =
  (271 * (get_u i) + 293 * (get_u j) + 283 * (get_u s)) mod 10000

let aut_gen_random (t : int) (n : int) (m : int) (d: int) : automate_t =
  let transitions = Array.make n [] in

  let mod_expr n m d i = Float.to_int ((Float.of_int (n * n * m)) /. (Float.of_int (d * (n - i + 1))) +. 1.0)
      in

  let rec make_transition_list base_state letter new_state buffer_list =
    if (base_state >= n) then ()
    else if (new_state >= n) then begin transitions.(base_state) <- buffer_list; make_transition_list (
        base_state + 1) 0 0 [] end
    else if (letter >= m) then make_transition_list base_state 0 (new_state + 1) buffer_list
    else if ( ((get_u (10 * t + (x base_state new_state letter))) mod (mod_expr n m d (base_state))) = 0 )
      then make_transition_list base_state (letter + 1) new_state ((letter, new_state)::buffer_list)
    else make_transition_list base_state (letter + 1) new_state buffer_list

```

```
  in make_transition_list 0 0 0 [];
  (n, m, transitions, [0], [n-1])


(* Question 2 *)
let q_2 (t : int) (n : int) (m : int) (d : int) : unit =
  let rand_aut = aut_gen_random t n m d in
  let trans_arr = aut_get_trans_arr rand_aut in

  let rec count_transitions count_buf cur_state trans_list = match trans_list with
    |[] -> if (cur_state >= n - 1) then count_buf else count_transitions count_buf (cur_state + 1) (trans_arr.(
        cur_state + 1))
    |h::t -> count_transitions (count_buf + 1) cur_state t

  in
  Printf.printf "   * Question 2 : A_%d(%d, %d, %d) contient %d transitions\n" t n m d (count_transitions 0
      (-1) [])
```

Question 3 + Oral 1 Notre définition des automates par un tableau des transitions nous permet de naturellement projeter nos automates sur un type graphe défini ci-dessous. Ainsi, nous pouvons appliquer l'algorithmie usuelle des graphes afin de trouver les états accessibles (via un parcours en profondeur).

```
type graphe_t = int * list_transition_t (* Graphe sous forme de Liste d'adjacence, 'pondere' par les lettres *)

let graphe_identity (g : graphe_t) : graphe_t = g

let graphe_get_n (g : graphe_t) : int = match g with
  |(n, t) -> n

let graphe_get_transitions (g : graphe_t) : list_transition_t = match g with
  |(n, t) -> t

let aut_to_graph (a : automate_t) : graphe_t = match a with
  |(q, sigma, delta, i, f) -> (q, delta)


let graphe_etats_accessibles_from_i (g : graphe_t) (i : int) : bool array =  (* Renvoie les etats accessibles
    depuis i *)
  let n, t = graphe_identity g in
  let tab_access = Array.make n false in

  let rec pp_recursif s =
    tab_access.(s) <- true;
    parcours_voisins_list (t.(s))

  and parcours_voisins_list l = match l with
      |[] -> ()
      |head::tail -> if(not tab_access.(transition_get_next head)) then (pp_recursif (transition_get_next head)
          ); parcours_voisins_list tail


  in pp_recursif i; tab_access


let aut_get_etats_accessibles (a : automate_t) : bool array =
    graphe_etats_accessibles_from_i (aut_to_graph a) 0

(* Question 3 *)

let q_3 (t : int) (n : int) (m : int) (d : int) =
```

```
  let rand_aut = aut_gen_random t n m d in
  let access_tab = aut_get_etats_accessibles rand_aut in
  Printf.printf "   * Question 3 : A_%d(%d, %d, %d) contient %d etats accessibles\n" t n m d (Array.fold_left (
      fun count b -> if (b) then (count + 1) else count) 0 access_tab)
```

Ainsi, notre fonction est de complexité $\mathcal{O}(n+|\delta|)$ (complexité usuelle du parcours en profondeur, sans compter les transformations non-nécessaires en graphe).

Questions 4, 5, 6 Il est possible de traiter simultanément les trois questions suivantes. Il apparaît en effet la notion d'intersection de langages, ce qui peut se calculer via l'automate produit. La question 4 revient à calculer le produit entre l'automate aléatoire et un automate reconnaissant ϕ. La question 5 se calcule grâce au produit entre deux automates aléatoires, et la question 6 se fait via le produit entre $k+1$ automates. (On remarquera qu'admettre une trace acceptante dans un langage donné revient à ce que l'intersection entre le langage reconnu et le langage donné soit non vide).

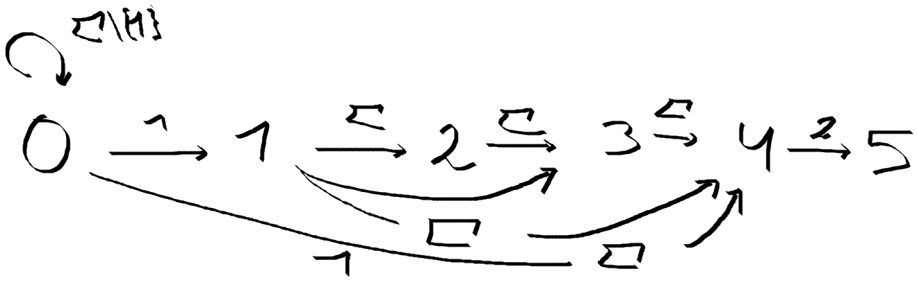
Ainsi :

```
let automate_phi_const : automate_t =
  (6, 10,
  [| [(0, 0); (1, 1); (1, 4); (2, 0); (3, 0); (4, 0); (5, 0); (6, 0); (7, 0); (8, 0); (9, 0)];
  ((List.init 10 (fun i -> (i, 2))) @ (List.init 10 (fun i -> (i, 3))) @ (List.init 10 (fun i -> (i, 4))));
  (List.init 10 (fun i -> (i, 3)));
  (List.init 10 (fun i -> (i, 4)));
  [(2, 5)];
  []|], [0], [5])


let list_prod_fun (la : 'a list) (lb : 'b list) (f : 'a -> 'a -> 'b): 'b list =
  let rec recursion a b = match (a, b) with
    |(_ , []) -> []
    |([], h::t) -> recursion la t
    |(ha::ta, hb::tb) -> (f ha hb)::(recursion ta b)
  in recursion la lb

let aut_get_voisins (a: automate_t) (q : etat_t) (l : alphabet_t) : etat_t list = match a with
  |(aq, sigma, delta, i, f) -> if(q >= aq || q < 0) then failwith "aut_get_voisins -> Cet etat est inconnu!"
  else let transition_list = delta.(q) in List.filter_map (fun (vl, n) -> if (vl <> l) then None else (Some n))
       transition_list

let aut_prod (a : automate_t) (b : automate_t) : automate_t = match (a, b) with
  |((aq, asigma, adelta, ai, af), (bq, bsigma, bdelta, bi, bf)) ->
    if(asigma <> bsigma) then failwith "aut_prod -> Les deux alphabets mis en jeu different!"
    else begin
      let new_q = aq * bq in
      let transition_tab = Array.make new_q [] in

      let rec build_transitions a_ind b_ind letter =
        if(a_ind >= aq) then ()
        else if(b_ind >= bq) then build_transitions (a_ind + 1) 0 0
        else if(letter >= asigma) then build_transitions a_ind (b_ind + 1) 0
        else begin
          let a_voisins = (aut_get_voisins a a_ind letter) in
          let b_voisins = (aut_get_voisins b b_ind letter) in

          let rec build_transitions_prod_list la lb build_list = match (la, lb) with
            |(_, []) -> transition_tab.(a_ind * bq + b_ind) <- (build_list @ transition_tab.(a_ind * bq + b_ind
                ))
            |([], h::t) -> build_transitions_prod_list a_voisins t build_list
            |(ha::ta, hb::tb) -> build_transitions_prod_list ta lb ((letter, (ha * bq + hb))::build_list)

          in build_transitions_prod_list a_voisins b_voisins []; build_transitions a_ind b_ind (letter + 1)
        end
```

```
      in build_transitions 0 0 0;
      (new_q, asigma, transition_tab, (list_prod_fun ai bi (fun a b -> a * bq + b)), (list_prod_fun af bf (fun a b -> a * bq + b)))
    end


let aut_is_language_empty (a : automate_t) : bool = match a with
  |(q, sigma, delta, i, f) -> let g = aut_to_graph a in

    let rec find_acceptant l accessibility_tab = match l with
      |[] -> false
      |ha::ta -> (accessibility_tab.(ha)) || (find_acceptant ta accessibility_tab) (* Evaluation parresseuse *)

    in let rec iter_initial l = match l with
      |[] -> false
      |h::t -> let access_tab = graphe_etats_accessibles_from_i g h in (find_acceptant f access_tab) || (iter_initial t)

  in not (iter_initial i) (* Not car afin de profiter de l'evaluation parresseuse, nous nous arretons lorsque vrai est trouve, ce qui correspond a un langage non vide*)

(* Question 4 *)

let q_4 n d t =

  let rec build_t_list t_val =
    if (t_val >= t+1) then []
    else begin
      let cur_automate = aut_gen_random t_val n 10 d in
        if(not (aut_is_language_empty (aut_prod cur_automate automate_phi_const)))
          then t_val::(build_t_list (t_val + 1)) else (build_t_list (t_val + 1))
      end

  in let rec find_min_card_list l = match l with
    |[] -> (Int.max_int, 0)
    |h::tail -> let m, s = (find_min_card_list tail) in if (h < m) then (h, s+1) else (m, s+1)


  in let t_list = build_t_list 1 in let min, size = find_min_card_list t_list in
  Printf.printf "   * Question 4 : Pour n = %d, d = %d, T = %d, Nous obtenons min = %d et Card = %d\n" n d t min size


(* Question 5 *)

let q_5 n d =
  let rec build_t_list t =
    if (t > 100) then []
    else begin
      let aut_a = aut_gen_random t n 10 d in let aut_b = aut_gen_random (t+1) n 10 d in
      if(not (aut_is_language_empty (aut_prod aut_a aut_b))) then t::(build_t_list (t+1)) else (build_t_list (t+1))
    end

  in let rec find_min_card_list l = match l with
    |[] -> (Int.max_int, 0)
    |h::tail -> let m, s = (find_min_card_list tail) in if (h < m) then (h, s+1) else (m, s+1)

  in let t_list = build_t_list 1 in let min, size = find_min_card_list t_list in
  Printf.printf "   * Question 5 : Pour n = %d, d = %d, Nous obtenons min = %d et Card = %d\n" n d min size

(* Question 6 *)
```

```
let q_6 k =

  let rec build_aut t k_param = if (k_param <= 0) then aut_gen_random t 10 2 5
  else (aut_prod (aut_gen_random t 10 2 5) (build_aut (t+1) (k_param-1)))

  in let rec build_t_list t =
    if (t > 10) then []
    else begin
      let aut = build_aut (5 * t) k in
      if(not (aut_is_language_empty (aut))) then t::(build_t_list (t+1)) else (build_t_list (t+1))
    end

  in let rec find_min_card_list l = match l with
    |[] -> (Int.max_int, 0)
    |h::tail -> let m, s = (find_min_card_list tail) in if (h < m) then (h, s+1) else (m, s+1)

  in let t_list = build_t_list 1 in let min, size = find_min_card_list t_list in
  Printf.printf "   * Question 6 : Pour k = %d, Nous obtenons min = %d et Card = %d\n" k min size
```

Question Oral 2 (TODO : A IMPLEMENTER EN TEX)



Question Oral 3 Comme nous l'avons dit précédemment, un algorithme convenant est celui de l'automate produit, qui consiste à créer un nouvel automate simulant une lecture simultanée dans les deux automates souhaités. La complexité d'une telle transformation est en théorie (avec les notations de la fonction, version naïve sans émonder) $\mathcal{O}(\Sigma \times |Q_a| \times |Q_b| \times |\delta_a| \times |\delta_b|)$.

0.2 Bisimulation

Question Oral 4 Montrons que la partition $R := \{\{0,1\},\{2\},\{3,4\}\}$ est une bisimulation (auquel cas les états 0,1 et 3,4 sont bisimilaires).

Notons $\mathcal{R} := \{(0,1),(1,0),(1,1),(0,0),(2,2),(3,3),(3,4),(4,3),(4,4)\}$ la relation associée à R.

Alors $\mathcal{R}$ est bien une relation d'équivalence :

— Il est immédiat que cette relation est réflexive par définition.
— La symétrie découle également de la définition de $\mathcal{R}$.
— Idem, la transitivité se vérifie car $\mathcal{R}$ est fini et est construit pour correspondre à la partition R

Tout couple (a,a) vérifie bien les deux conditions données (la première est évidente). En effet, si $a \neq 2$, alors avec $a = 0$ ou $a = 1$, il suffit de choisir $p = p' = 2$ pour $\sigma = 0$ ($\sigma = 1$ impossible sur ce couple). Idem pour $a = 3$ et $a = 4$.

Pour $(2,2)$, avec $\sigma = 0$, le couple $p = p' = 1$ convient, et avec $\sigma = 1$, le couple $p = p' = 4$ convient.

Les deux seuls couples restants sont $(0,1)$ et $(3,4)$ (et leur permutation, ce qui n'importe pas). Pour ces deux couples, la première condition est vérifiée.

Pour le premier couple, avec $\sigma = 0$ (le cas $\sigma = 1$ est impossible), prenons $p = p' = 2$. Alors $(2,2) \in \mathcal{R}$ et $(0,0,2) \in \delta$, et $(1,0,2) \in \delta$. Idem pour le deuxième couple avec $\sigma = 1$.

$\mathcal{R}$ est bien une Bisimulation pour cet automate, or, $(0,1) \in \mathcal{R}$ et $(3,4) \in \mathcal{R}$. D'où le fait que 0,1 et 3,4 soient bisimilaires.

Question 7 L'implémentation de l'algorithme proposé ne pose pas de difficulté particulière :

```
type state_partition_t = etat_t list list

let list_is_empty (l : 'a list) : bool = match l with |[] -> true |_ -> false
let rec list_contains (l : 'a list) (e : 'a) : bool = match l with
  |[] -> false
  |h::t -> (e = h) || (list_contains t e)

let list_filter_out (l : 'a list) (filter : 'a list) : 'a list =
  if(list_is_empty filter) then l else  (* PLus rapide sur filtre vide *)
  let rec recursion la = match la with
    |[] -> []
    |h::t -> if(list_contains filter h) then (recursion t) else h::(recursion t)
  in recursion l

let list_split_on (l : 'a list) (filter : 'a list) : ('a list * 'a list) =
  if(list_is_empty filter) then ([], l) else  (* PLus rapide sur filtre vide *)
  let rec recursion la = match la with
    |[] -> ([], [])
    |h::t -> let inside, outside = (recursion t) in if (list_contains filter h) then ((h::inside), outside)
        else (inside, (h::outside))
  in recursion l

let aut_trivial_partition (a : automate_t) : state_partition_t = [(List.init (aut_get_states a) (fun i -> i))]
let aut_basic_nerode_partition (a : automate_t) : state_partition_t = match a with
  |(q, sigma, delta, i, f) -> [list_filter_out (List.init (aut_get_states a) (fun i -> i)) f; f]

let aut_predecessor (a : automate_t) (state_list: etat_t list) (letter : alphabet_t) : etat_t list = match a
    with
  |(q, sigma, delta, i, f) ->
    let rec build_predecessor_list state build_list =
      if(state >= q) then build_list
```

```
      else begin
        let voisins = aut_get_voisins a state letter in
        let in_v, out_v = list_split_on voisins state_list in
        if (not (list_is_empty in_v)) then build_predecessor_list (state + 1) (state::build_list)
        else build_predecessor_list (state + 1) build_list
      end
    in build_predecessor_list 0 []


let aut_algorithme_5 (a : automate_t) : state_partition_t = match a with
  |(q, sigma, delta, i, f) ->
    let base_partition = aut_basic_nerode_partition a in

    let rec find_letter_class_to_split_on cur_class partition letter = match partition with
      |[] -> ([], [])
      |h::t -> begin if(letter >= sigma) then (find_letter_class_to_split_on cur_class t 0)
        else begin
          let in_class, out_class = list_split_on cur_class (aut_predecessor a h letter) in
          if( (not (list_is_empty in_class)) && (not (list_is_empty out_class)) ) then (in_class, out_class)
          else find_letter_class_to_split_on cur_class partition (letter + 1)
        end
      end


    in let rec find_suitable_classes current_partition =
      let rec iter_through_partitions cur_part previous_list = match cur_part with
        |[] -> current_partition
        |h::t -> let in_class, out_class = find_letter_class_to_split_on h current_partition 0 in
        if(not (list_is_empty in_class)) then (find_suitable_classes (in_class::out_class::(previous_list @ t))
            ) else (iter_through_partitions t (h::previous_list))
      in iter_through_partitions current_partition []
    in find_suitable_classes base_partition


(* Question 7 *)

let q_7 t n m d =
  Printf.printf "   * Question 7 : La partition de A_%d(%d, %d, %d) donne %d classes d'equivalence\n" t n m d (
      List.length (aut_algorithme_5 (aut_gen_random t n m d)))
```

Question Oral 5 Remarquons premièrement le fait qu'à toute étape de l'algorithme, $\mathcal{P}$ contient une partition de Q : Par récurrence sur le nombre d'appels de la boucle :

Ceci est vrai initialement, car $\mathcal{P} = \{Q \setminus F, F\}$.

Si ceci est vrai après n appels de la boucle, alors lors du $n+1$è appel, $\mathcal{P}$ reçoit $(\mathcal{P} \setminus P) \cup (P \cap \text{Pred}) \cup (P \setminus \text{Pred})$.

Ainsi, en considérant $p_1, \dots, p_k$ les éléments de $\mathcal{P} \setminus P$, par hypothèse de récurrence, $\forall j \in [\![1, k]\!]$, $p_j \cap P = \emptyset$.

Or, $(P \cap \text{Pred}) \subset P$ et $(P \setminus \text{Pred}) \subset P$, donc ceci garantit bien la séparation des ensembles contenus dans $\mathcal{P}'$ ($\mathcal{P}$ après l'appel de boucle). Or, l'union $(P \cap \text{Pred}) \cup (P \setminus \text{Pred})$ donne P, et $p_1 \cup \dots \cup p_k \cup P = Q$, donc $\mathcal{P}'$ est bien une partition de Q.

Remarquons de plus qu'à chaque appel de boucle, $|\mathcal{P}|$ augmente strictement (de 1). Or, une partition de Q ne peut posséder plus de $|Q|$ éléments. Il ne peut alors pas y avoir plus de $|Q|$ appels de boucle. L'algorithme termine donc.

La relation de récurrence associée à la partition renvoyée par l'algorithme est de plus une bisimulation : Soit $\mathcal{P} = \{p_i \mid i \leq k\}$ la partition renvoyée.

L'algorithme renvoie $\mathcal{P}$. Alors il n'existe aucun couple $(P,P') \in \mathcal{P}^2$ et aucune lettre $\sigma \in \Sigma$ tels que $P \cap (\text{Pred}(P',\sigma)) \neq \emptyset$ et $P \not\subset \text{Pred}(P',\sigma)$.

Remarquons que la première condition est automatiquement vérifiée par construction : L'algorithme prend comme partition initiale $\{Q \setminus F, F\}$. Or, $\mathcal{P}$ provient de raffinement de ces ensembles (les ensembles de $\mathcal{P}$ sont tous des sous-ensembles de ces deux ensembles). Ainsi, il n'est pas possible d'avoir un p_i contenant à la fois des états de $Q \setminus F$ et de F.

Si la deuxième condition n'est pas respectée : Alors il existe un couple $(a,b) \in (p_i)^2$ pour un certain $i \leq k$, tel qu'il existe une lettre $\sigma \in \Sigma$, telle que pour tout $p, p' \in Q$ avec $(a,\sigma,p) \in \delta$ et $(b,\sigma,p') \in \delta$, alors $(p,p') \notin \mathcal{R}$ (i.e n'appartiennent pas au même p_j).

Alors, en prenant $P = p_i$, $P' = p_j$ contenant p', et σ la lettre réalisant la proposition précédente, par définition, $P \cap \text{Pred}(P',\sigma) \neq \emptyset$ car contient b, et $P \not\subset \text{Pred}(P',\sigma)$ car $\text{Pred}(P',\sigma)$ ne contient pas a. D'où l'absurdité par hypothèse de terminaison.

Dès lors, pour tout couple $(a,b) \in \mathcal{R}$, ce couple vérifie les deux conditions faisant que $\mathcal{R}$ est une bisimulation.

Question Oral 6 Soient $\mathcal{A}$ et $\mathcal{B}$, deux automates bisimilaires. Alors $\mathcal{U}$ est tel que q_0 et q'_0 sont bisimilaires.

Par récurrence sur la taille d'un mot $w \in \Sigma^n$, montrons que q_w et q'_w sont bisimilaires (où q_w et q'_w désignent les états atteints par la lecture de w dans $\mathcal{A}$ (respectivement $\mathcal{B}$) :

q_0 et q'_0 étant bisimilaires, ceci est vrai pour le mot vide.

Supposons donc que pour tout mot w de taille $\leq n$, alors q_w et q'_w sont bisimilaires.

Alors pour un mot $\tilde{w}$ de taille $(n+1)$, par définition, $\exists \sigma \in \Sigma,\ w \in \Sigma^n,\ \tilde{w} = w\sigma$. Donc par hypothèse de récurrence, $q_w \sim q'_w$, et de ce fait, il existe $p, p' \in Q$ tels que $p \sim p'$, avec $(q_w,\sigma,p) \in \delta$ et $(q'_w,\sigma,p') \in \delta$. Ainsi, $p \sim p'$, et $p = q_{\tilde{w}}$, $p' = q'_{\tilde{w}}$. Ce qui conclut bien la récurrence.

En particulier, pour tout mot $w \in \mathcal{L}(\mathcal{A})$, alors $q_w \in F \iff q'_w \in F$, donc $w \in \mathcal{L}(\mathcal{B})$: Deux automates Bisimilaires reconnaissent bien le même langage.

Réciproquement, si les deux automates $\mathcal{A}$ et $\mathcal{B}$ reconnaissent le même langage, avec $\mathcal{A}$ et $\mathcal{B}$ déterministes complets.

Construisons la bisimulation $\mathcal{R}$, donnée par la clôture Réflexive Symétrique Transitive de la relation donnée par :

— $q_0 \sim q'_0$.
— $\forall q \in Q,\ q' \in Q',\ q \sim q' \implies \forall \sigma \in \Sigma,\ \text{Succ}(q,\sigma) \sim \text{Succ}(q',\sigma)$ (le successeur existe car les automates considérés sont déterministes et complets.)

Alors $\mathcal{R}$ est bien une relation d'équivalence, car est symétrique, réflexive et transitive par définition.

De plus, $\mathcal{R}$ constitue bien une bisimulation : On montre par récurrence sur la longueur de tout mot que $q_w \sim q'_w$ (toujours avec les même notations) :

Ceci est vrai par définition pour le mot vide. Soit donc un mot $\tilde{w} \in \Sigma^{n+1}$, en supposant la propriété respectée par tout mot de taille $\leq n$.

Alors $\tilde{w} = w'\sigma$ pour $w \in \Sigma^n$, et $\sigma \in \Sigma$. Donc, par H.R, nous avons $q_w \sim q'_w$. Ainsi, par définition, $q_{\tilde{w}} \sim q'_{\tilde{w}}$.

Ceci garantit alors par définition la première propriété : Si $q \sim q'$, avec $q \in Q$ et $q' \in Q'$, alors $q \in F \iff q' \in F'$, car $\mathcal{A}$ et $\mathcal{B}$ reconnaissent le même langage. et si $q' \in Q$, il en va de même car une seule lecture de $\tilde{w}$ est possible : Il est impossible que $\tilde{w}$ soit reconnu dans $\mathcal{A}$ et soit rejeté dans $\mathcal{A}$ en même temps. (car le fait que deux sommets d'un même automate soient équivalents provient de la clôture transitive de $, donc nous nous ramenons au premier cas.$)

La deuxième propriété est de plus vérifiée par définition de $\mathcal{R}$.

Ainsi, $\mathcal{R}$ est bien une bisimulation sur $\mathcal{U}$, telle que $q_0 \sim q'_0$, donc les deux automates $\mathcal{A}$ et $\mathcal{B}$ sont bien bisimilaires.

En revanche, en considérant un automate non déterministe complet $\mathcal{A}$ et son automate déterministe associé $\mathcal{B}$, nous pouvons remarquer que la facilité de concevoir un automate non déterministe vient du fait que $\mathcal{A}$ et $\mathcal{B}$ ne sont en général pas équivalents. TODO : Trouver un contre-exemple.

Questions 8, 9 La question 8 consiste à remplacer les transitions souhaitées, en supprimant néanmoins les doublons pouvant apparaître. La question 9 nécessite l'implémentation de l'automate union, il suffit alors d'appliquer l'algorithme donné à la question oral 5 afin de déterminer une bisimilarité sur cet automate union, et donc d'en déduire si les deux automates mis en jeu sont bisimilaires.

```
let rec list_insert_sans_doublon (l :'a list) (e : 'a) : 'a list = match l with
  |[] -> [e]
  |h::t -> if(h = e) then l else h::(list_insert_sans_doublon t e)

let y (i : etat_t) (j : etat_t) (lettre: alphabet_t) = 41 * i + 31 * j + lettre


let aut_gen_random_errone (t : int) (n : int) (m : int) (d : int) (p : int) =
  let a_base = aut_gen_random t n m d in match a_base with
    |(q, sigma, delta, i, f) ->
      let new_transitions = Array.make q [] in


      (* Attention, il faut cette fois retirer les doublons eventuels !*)

      let rec build_transitions cur_state delta_list new_list = match delta_list with
        |[] -> new_transitions.(cur_state) <- new_list
        |head::tail -> let letter, next = transition_identity head in
          if((get_u (5 * t + (y cur_state next letter))) mod p = 0) then
            build_transitions cur_state tail (list_insert_sans_doublon new_list (((get_u (5*t + (y cur_state
                next letter) + 100)) mod m), next))
          else build_transitions cur_state tail (list_insert_sans_doublon new_list head)

      in let rec iterate_state s = if (s >= q) then () else begin build_transitions s (delta.(s)) [];
          iterate_state (s+1) end
      in iterate_state 0; (q, sigma, new_transitions, i, f)

(* Question 8 *)

let q_8 (t : int) (n : int) (m : int) (d : int) (p : int) : unit =
  let rand_aut = aut_gen_random_errone t n m d p in
  let trans_arr = aut_get_trans_arr rand_aut in

  let rec count_transitions count_buf cur_state trans_list = match trans_list with
    |[] -> if (cur_state >= n - 1) then count_buf else count_transitions count_buf (cur_state + 1) (trans_arr.(
        cur_state + 1))
    |h::t -> count_transitions (count_buf + 1) cur_state t

  in
  Printf.printf "   * Question 8 : A^{err(%d)}_%d(%d, %d, %d) contient %d transitions\n" p t n m d (
      count_transitions 0 (-1) [])
```

```
38
39
40  let aut_offset (a : automate_t) (s : int) : automate_t = match a with
41    |(q, sigma, delta, i, f) ->
42    let na = Array.init q (fun l -> (List.map (fun (letter, next) -> (letter, next + s)) (delta.(l)))) in
43      (q + s, sigma, na, (List.map (fun k -> k+s) i), (List.map (fun k -> k+s) f))
44
45  let aut_union (a : automate_t) (b : automate_t) : automate_t = match (a,b) with
46    |((aq, asigma, adelta, ai, af), (bq, bsigma, bdelta, bi, bf)) ->
47      let new_b = aut_offset b aq in
48      (aq + bq,
49      (max asigma bsigma),
50      (Array.init (aq + bq) (fun i -> if (i < aq) then adelta.(i) else (aut_get_trans_arr new_b).(i - aq))),
51       ai @ (aut_get_init_state new_b),
52       af @ (aut_get_fin_state new_b))
53
54
55  (* Question 9 *)
56
57  let q_9 (n : int) (m : int) (d : int) (p : int) : unit =
58
59    let rec find_class_of (l : state_partition_t) (a : etat_t) : etat_t list = match l with
60      |[] -> failwith "find_class_of -> La partition ne contient pas l'etat souhaite"
61      |h::t -> if(list_contains h a) then h else (find_class_of t a)
62
63    in let partition_same_class (l : state_partition_t) (a : etat_t) (b : etat_t) : bool =
64      list_contains (find_class_of l a) b
65
66    in let rec build_t_list cur_t =
67      if (cur_t > 100) then []
68      else begin
69        let rand_aut = aut_gen_random cur_t n m d in
70        let rand_aut_error = aut_gen_random_errone cur_t n m d p in
71        let union = aut_union rand_aut rand_aut_error in
72        let union_partition = aut_algorithme_5 union in
73        if(partition_same_class union_partition 0 (aut_get_states rand_aut)) (* Les automates consideres pour l'
              union ne possedent qu'un unique etat d'entree par hypothese *)
74          then cur_t::(build_t_list (cur_t + 1)) else build_t_list (cur_t + 1)
75        end
76    in let rec find_min_card_list l = match l with
77      |[] -> (Int.max_int, 0)
78      |h::tail -> let m, s = (find_min_card_list tail) in if (h < m) then (h, s+1) else (m, s+1)
79
80    in let t_list = build_t_list 1 in let min, size = find_min_card_list t_list in
81    Printf.printf "   * Question 9 : Pour p = %d, n = %d, m = %d, d = %d, Nous obtenons min = %d et Card = %d\n"
        p n m d min size
```

0.3 Équivalence de traces

Question Oral 7 La manière usuelle (naïve) de déterminer si $\mathcal{L}(\mathcal{A}) \subset \mathcal{L}(\mathcal{B})$ revient à remarquer que cette inclusion est vraie si et seulement si $\mathcal{L}(\mathcal{A}) \cap (\mathcal{L}(\mathcal{B}))^c = \emptyset$, ce que nous pouvons déterminer en vérifiant si l'un des états terminaux est accessible.

Or, le langage $(\mathcal{L}(\mathcal{B}))^c$ est calculable en supposant $\mathcal{B}$ déterministe complet : Il suffit de poser $F' \leftarrow F^c$ (le complémentaire des états terminaux).

En toute généralité, il faudrait alors déterminiser et compléter l'automate $\mathcal{B}$, ce qui est en théorie en complexité exponentielle en la taille de Q_B. En pratique, $|Q_B| \leq 13$ est réalisable, ce qui réduit grandement l'application naïve d'un tel résultat.