Corrigé du TP sur la déterminisation des automates

Préliminaires

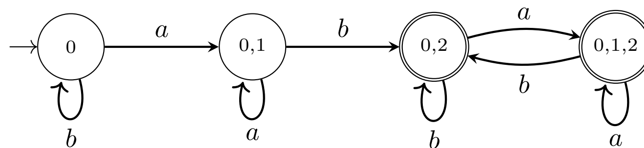
On obtient facilement :

```
let appartient i q = Int.logand (Int.shift_left 1 i) q <> 0 ;;

let intersecte q1 q2 = Int.logand q1 q2 <> 0 ;;

let ajoute i q = Int.logor (Int.shift_left 1 i) q ;;
```

Partie I : les automates déterministes

1) Pour $\mathcal{A}_1$, on obtient, en choisissant $0 = \{0\}$, $1 = \{0, 1\}$, $2 = \{0, 2\}$ et $3 = \{0, 1, 2\}$:

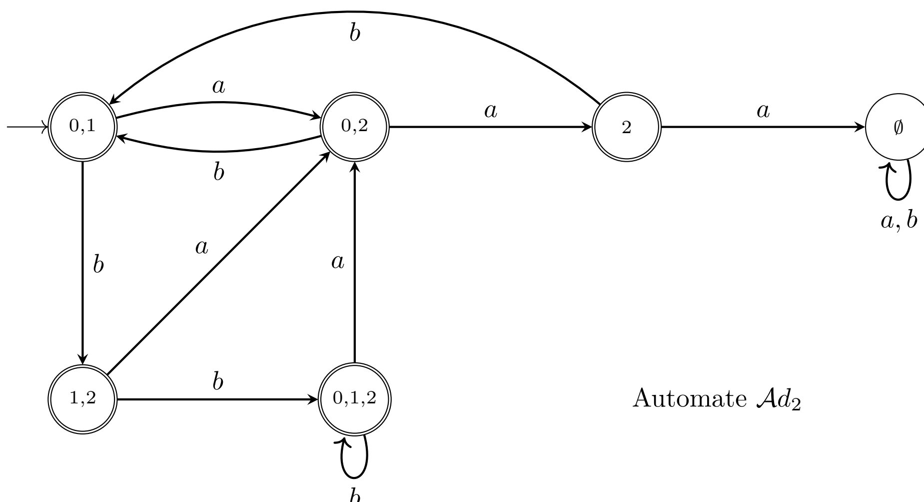
```
let a1d = {m = 4; i0 = 0; fin = 12;
                      delta = [|[|1;0|];[|1;2|];[|3,2|];[|3;2|]|]}
```



Automate $\mathcal{A}d_1$

Pour $\mathcal{A}_2$, en choisissant $0 = \{0, 1\}$, $1 = \{0, 2\}$, $2 = \{1, 2\}$, $3 = \{0, 1, 2\}$, $4 = \{2\}$ et $5 = \emptyset$, on obtient :

```
let a2d = {m = 6; i0 = 0; fin = 31;
                      delta = [|[|1;2|];[|4;0|];[|1;3|];[|1;3|];[|5;0|];[|5; 5|]|]}
```



Automate $\mathcal{A}d_2$

2) On va parcourir le mot en avançant dans l'automate, en arrêtant le calcul si l'on atteint l'état m.

```
let deplacement aut q u =
   let etat = ref q and i = ref 0 in
   while !etat != aut.m && !i < String.length u do
      match u.[!i] with
         | 'a' -> etat := aut.delta.(!etat).(0); incr i;
         | _ -> etat := aut.delta.(!etat).(1); incr i;
   done;
   !etat;;
```

3) La fonction `reconnu_det` est alors élémentaire :

```
let reconnu_det aut u = let j = deplacement aut (aut.i0) u in
   if j = aut.m then
      false
   else
      appartient j aut.fin;;
```

Partie II : les automates non déterministes

1) On écrit directement :

```
let a1 = {n=3; ini = 1; fin = 4; a = [|3;0;4|]; b = [|1;4;4|]};;
let a2 = {n=3; ini = 3; fin = 5; a = [|4;5;0|]; b = [|2;4;3|]};;
```

Pour définir plus facilement les automates, on peut utiliser la fonction d'initialisation ci-dessous, où les paramètres li, lf, la et lb sont respectivement les listes des états initiaux, des états finals, des transitions d'étiquette a eta des transitions d'étiquette b :

```
let init(n,li,lf,la,lb)=
   let i = ref 0 and f = ref 0 in
   List.iter (fun a -> i := ajoute a !i) li;
   List.iter (fun a -> f := ajoute a !f) lf;
   let aut = {n = n; ini = (!i); fin = (!f) ; a = Array.make n 0; b = Array.make n 0} in
   List.iter (fun (q1,q2) -> aut.a.(q1) <- ajoute q2 aut.a.(q1)) la;
   List.iter (fun (q1,q2) -> aut.b.(q1) <- ajoute q2 aut.b.(q1)) lb;
   aut;;
```

2) On initialise q' à la valeur 0 (i.e. à l'ensemble vide) puis, pour chaque élément i appartenant à q, on fait la réunion de q' et de $g.(i)$. Cela donne :

```
let calcul g q = let n = Array.length g and qprime = ref 0 in
   for i=0 to n-1 do
      if appartient i q then
         qprime := Int.logor (!qprime) g.(i)
   done;
   !qprime;;
```

3) On part de l'ensemble des états initiaux puis on lit les lettres du mot u : il reste à tester si l'ensemble atteint après la lecture intersectée l'ensemble des états finals.

```
let reconnu aut u = let q = ref aut.ini in
   for k = 0 to (String.length u)-1 do
      let g = match u.[k] with 'a'-> aut.a | _ -> aut.b in
      q := calcul g !q
   done;
   intersecte !q aut.fin;;
```

4) Nous utilisons ici une fonction qui, appliquée à un graphe g et à un ensemble d'états q de $\mathbb{N}_n$, renvoie l'ensemble des états j tels qu'il existe i dans q tel que (i, j) est un arc de g :

```
let nouveaux_etats g q n = let qprime = ref q in
   for i=0 to n-1 do
      if appartient i q then
         qprime := Int.logor (!qprime) g.(i)
   done;
   !qprime;;
```

On part ensemble de l'ensemble des états initiaux puis on applique la fonction précédente pour propager l'exploration du graphe (le graphe g utilisé est la réunion des graphes d'étiquettes a et b), en arrêtant le calcul dès que l'on ne découvre pas de nouveaux états.

```
let accessibles aut =
   let g = Array.make aut.n 0 in
   for i=0 to aut.n-1 do
      g.(i) <- Int.logor aut.a.(i) aut.b.(i)
   done;
   let rec propagation q = match nouveaux_etats g q aut.n with
      | qprime when q <> qprime -> propagation qprime
      | _ -> q in
   propagation aut.ini;;
```

Pour les états co-accessibles, on commence par calculer le « graphe inverse », i.e. le graphe contenant les couples (i, j) tels que (j, a, i) ou (j, b, i) est une transition de l'automate. La suite du calcul est identique, en partant de l'ensemble des états finals.

```
let calcul_graphe_inverse aut = let g = Array.make aut.n 0 in
   for i=0 to aut.n-1 do
      for j=0 to aut.n-1 do
         if appartient i aut.a.(j) || appartient i aut.b.(j) then
            g.(i) <- ajoute j g.(i)
      done;
   done;
   g;;

let co_accessibles aut =
   let g = calcul_graphe_inverse aut in
   let rec aux q = match nouveaux_etats g q aut.n with
      | qprime when q <> qprime -> aux qprime
      | _ -> q in
   aux aut.fin;;
```

Partie III : automate des parties et détermination d'un automate

1) Il suffit d'utiliser la fonction `calcul` :

```
let automate_parties aut =
   let m = Int.shift_left 1 aut.n in
   let f = ref 0 in
   let delta=Array.make_matrix m 2 0 in
   for q=0 to m-1 do
      delta.(q).(0) <- calcul aut.a q; delta.(q).(1) <- calcul aut.b q;
      if intersecte q aut.fin then f := ajoute q (!f)
   done;
   {m = m; i0 = aut.ini; fin = !f; delta = delta} ;;
```

2) On utilise :

- une table de hachage t pour numéroter les états au fur et à mesure de leur découverte ;
- une référence m qui pointe vers le nombre d'états déjà rencontrés ;
- une liste $trans$ de triplets permettant de construire la fonction de transition ;
- une pile p contenant les états découverts mais dont les successeurs n'ont pas encore été calculés ;
- une référence $final$ qui code l'ensemble des états finals déjà rencontrés.

Dès qu'un nouvel état q est découvert, on définit dans la table t l'entrée $(q, !m)$, on ajoute (si q contient un état final) q à $final$ et on incrémente m. Tant que la pile est non vide, on dépile l'élément q qui est en tête, on calcule les états $qa = q.a$ et $qb = q.b$: s'ils n'ont pas encore été découverts, on leur donne un numéro et on les ajoute à la pile ; comme les états q, qa, qb ont maintenant des numéros m_0, m_1, m_2, que l'on retrouve dans la table t, on ajoute à la liste $trans$ le triplet (m_0, m_1, m_2). Une fois que la liste est vide, tous les états accessibles ont été atteints : il reste à définir l'automate déterministe.

```
let determiniser aut =
   let m = ref 1 in
   let trans = ref [] and final = ref 0 and p = ref [aut.ini] and t = Hashtbl.create (2*aut.n) in
   Hashtbl.add t aut.ini 0 ;
   if intersecte aut.ini aut.fin then
      final := 1;
   while !p <> [] do
      let q = List.hd !p in
      p := List.tl !p;
      let qa,qb = calcul aut.a q,calcul aut.b q in
      if not (Hashtbl.mem t qa) then
         (Hashtbl.add t qa (!m);
         if intersecte qa aut.fin then final := ajoute (!m) (!final);
         incr m;
         p := qa::(!p)) ;
      if not (Hashtbl.mem t qb) then
         (Hashtbl.add t qb (!m);
         if intersecte qb aut.fin then final := ajoute (!m) (!final);
         incr m;
         p := qb::(!p)) ;
      trans := (Hashtbl.find t q,Hashtbl.find t qa,Hashtbl.find t qb)::(!trans);
   done;
 let delta = Array.make_matrix !m 2 0 in
    List.iter (fun (q,qa,qb) -> delta.(q).(0) <- qa; delta.(q).(1) <- qb) (!trans) ;
   {m = !m; i0 = 0; fin = !final; delta = delta} ;;
```