Devoir surveillé n° 04 — corrigé

Q 1) Tout mot est synchronisant pour une machine à un seul état.

Q 2) Remarquons que si n est pair, $\delta^*(q_0, a^n) = q_0 \neq q_1 = \delta^*(q_1, a^n)$ et que si n est impair $\delta^*(q_0, a^n) = q_1 \neq q_0 = \delta^*(q_1, a^n)$. Ainsi, la machine M_1 n'admet aucun mot synchronisant.

Q 3) Le mot bcb est un mot synchronisant de trois lettres qui mène vers l'état q_3.

Q 4) Il faut bien penser à allouer et initialise chaque sous-tableau.

C
```c
machine* init_machine(int nb_etats) {
  machine* m = malloc(sizeof(machine));
  m->nb_etats = nb_etats;
  m->delta = malloc(nb_etats * sizeof(int*));
  for (int q = 0; q < nb_etats; q++) {
    m->delta[q] = malloc(nb_lettres * sizeof(int));
    for (int c = 0; c < nb_lettres; c++) {
      m->delta[q][c] = q;
    }
  }
  return m;
}
```

Q 5) Il ne faut pas oublier de commencer par libérer chaque sous-tableau.

C
```c
void liberer_machine(machine* m) {
  for (int q = 0; q < m->nb_etats; q++) {
    free(m->delta[q]);
  }
  free(m->delta);
  free(m);
}
```

Q 6) Pas de difficulté.

C
```c
int delta_etoile(machine* m, int q, char* u) {
  int i = 0;
  while (u[i] != '\0') {
    q = m->delta[q][(int)u[i]];
    i++;
  }
  return q;
}
```

Q 7) On vérifie simplement que toute lecture à partir de tout état mène vers le même état. Par hypothèse il y a bien au moins un état (même deux).

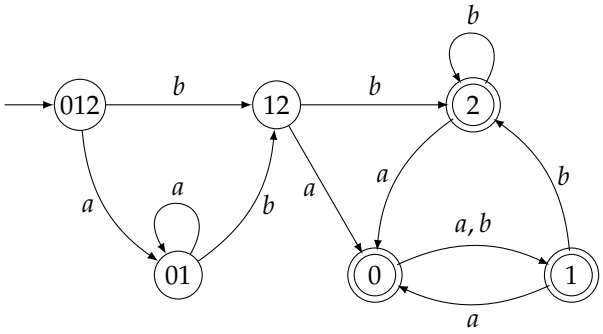
C
```c
bool est_synchronisant(machine* m, char* u) {
  int qs = delta_etoile(m, 0, u);
  for (int q = 1; q < m->nb_etats; q++) {
    if (delta_etoile(m, q, u) != qs) {return false;}
  }
  return true;
}
```

Q 8) Soit M une machine admettant un mot sychronisant $u \in \Sigma^*$ et supposons que pour tout $q \neq q'$ et $a \in \Sigma$, $\delta(q,a) \neq \delta(q',a)$. Soit $q \neq q'$ deux états distincts (on a supposé que toute machine possède au moins deux états). Montrons par induction que pour tout $v \in \Sigma^*$ on a $\delta^*(q,v) \neq \delta^*(q',v)$. C'est évident si $v = \epsilon$. Si $v = v'a$, par hypothèse d'induction on a $\delta^*(q,v') \neq \delta^*(q',v')$ donc $\delta^*(q,v) = \delta(\delta^*(q,v'),a) \neq \delta(\delta^*(q',v'),a) = \delta^*(q',v)$. Or $\delta^*(q,u) = \delta^*(q',u)$ car u est synchronisant. On conclut par l'absurde.

Q 9) S'il existe un mot synchronisant u pour M qui synchronise vers un état q_0, alors $\widehat{\delta}^*(Q,u) = \{q_0\}$. On en déduit qu'il existe un mot synchronisant pour M si et seulement si un singleton est accessible depuis l'état $Q \in \widehat{Q}$ dans $\widehat{M}$.

Q 10) On définit l'automate déterministe $(\widehat{Q}, \Sigma, Q, \widehat{T}, \widehat{\delta})$ avec $\widehat{T} = \{\{q\} \mid q \in Q\}$. D'après la question précédente, cet automate reconnaît bien l'ensemble de tous les mots synchronisants de M. On en déduit que $LS(M)$ est reconnaissable.

Q 11) On détermine l'automate des parties. On ne représente que les états utiles (l'état initial étant Q). Notons qu'on peut simplifier cet automate en fusionnant 0, 1 et 2 d'une part et 012 et 01 d'autre part.



Q 12) Soit $M = (Q, \Sigma, \delta)$ une machine et q_0 un état de M. On pose $M' = (Q, \Sigma, \delta')$ avec δ' définie par :

- $\forall q \in Q \setminus \{q_0\}, \forall a \in \Sigma, \delta'(q, a) = \delta(q, a)$;
- $\forall a \in \Sigma, \delta'(q_0, a) = q_0$.

On remarque qu'avec cette construction, si u est un mot synchronisant pour M', alors $\forall q \in Q, \delta'^*(q, u) = q_0$. On en déduit que pour tout état q le chemin dans la machine M menant de q à $\delta^*(q, u)$ passe forcément par q_0. Réciproquement, si pour tout état q le chemin dans la machine M menant de q à $\delta^*(q, u)$ passe forcément par q_0, alors u est synchronisant dans la machine M'.

Q 13) On effectue simplement un parcours, par exemple en profondeur. Le seul petit piège est que la liste de voisins est une liste de couples et non de sommets.

OCaml
```
let accessibles g depart =
  let n = Array.length g in
  let vu = Array.make n false in
  let nb_accessibles = ref 0 in
  let rec explorer s =
    vu.(s) <- true;
    incr nb_accessibles;
    let f (v, _) = if not vu.(v) then explorer v in
    List.iter f g.(s)
  in
  let f s = if not vu.(s) then explorer s in
  List.iter f depart;
  !nb_accessibles
```

Q 14) Un appel à `explorer s` n'est possible que si `s` n'est pas marqué. Or un sommet est directement marqué lors d'un appel à `explorer s`. Il ne peut donc y avoir au plus qu'un seul appel à explorer par sommet $s \in S$ qui sont en nombre finis. Il n'y a pas d'autre éventuel problème de non terminaison.

Q 15) La complexité est en $O(|S| + |A|)$.

Q 16) On a nécessairement $|\delta^*(Q, u_0)| \geq |\delta^*(Q, u_1)| \geq \cdots \geq |\delta^*(Q, u_r)| = 1$. En effet la machine étant déterministe, en lisant un mot depuis un état, on ne peut atteindre qu'un seul état. En lisant un mot depuis un ensemble d'états, le nombre d'états atteignables ne peut que diminuer (en cas de collisions). De plus, comme $u = u_r$ est synchronisant, $|\delta^*(Q, u_r)| = 1$.

Q 17) Le sens direct est évident, car $u_{q,q'} = u$ convient (u est synchronisant). Réciproquement, définissons les suites :

- $u_0 = v_0 = \varepsilon$ et $Q_0 = Q$;
- pour $i \in \mathbb{N}$, si $|Q_i| = 1$, alors on s'arrête, sinon, il existe deux états $q \neq q'$ de Q_i et donc un mot $u_{i+1} = u_{q,q'}$ qui synchronise q et q'. On pose alors $Q_{i+1} = \delta^*(Q_i, u_{i+1})$ et $v_{i+1} = v_i u_{i+1}$. On a $|Q_{i+1}| = |\delta^*(Q_i, u_{i+1})| < |Q_i|$.

La suite des $(|Q_i|)$ est ainsi strictement décroissante et minorée par 1, donc à partir d'un rang $j < n$, on a $|Q_j| = 1$. Le mot v_j est donc synchronisant puisque $Q_j = \delta^*(Q_0, v_j)$ par définition des suites et de δ^*.

Q 18) Il y a $\tilde{n} = \binom{n}{1} + \binom{n}{2} = n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2}$ parties à 1 ou 2 éléments de Q.

Q 19) On remarque que $\{[\![\frac{q(q+1)}{2}, \frac{q(q+1)}{2} + q]\!], q \in [\![0, n-1]\!]\}$ forme une partition de $[\![0, \tilde{n} - 1]\!]$. On en déduit que la fonction φ_n telle que $\varphi_n(\{q\}) = \frac{q(q+1)}{2}$ et $\varphi_n(\{q_1, q_2\}) = \frac{q_1(q_1+1)}{2} + q_2 + 1$ si $q_1 > q_2$ est bien une bijection convenable.

OCaml
```
let rec phi n = function
  | [q] -> (q * (q + 1)) / 2
  | [q1; q2] when q1 > q2 -> phi n [q1] + q2 + 1
  | [q1; q2] when q1 < q2 -> phi n [q2 ; q1]
  | _ -> assert false
```

Q 20) On transforme l'état de $\widetilde{M}$ en la liste d'un ou deux états de M, on applique δ en distinguant les cas et on retransforme en état de $\widetilde{M}$.

OCaml
```
let delta_tilde m q a =
  let n = Array.length m in
  match phi_inv n q with
  | [q] -> phi n [m.(q).(a)]
  | [q1; q2] when m.(q1).(a) = m.(q2).(a) ->
      phi n [m.(q1).(a)]
  | [q1; q2] -> phi n [m.(q1).(a); m.(q2).(a)]
  | _ -> assert false
```

Q 21) On parcourt tous les états entre 0 et $\tilde{n} - 1$ et toutes les lettres pour déterminer quelles transitions mènent à un état de $\widetilde{M}$ donné, afin de mettre à jour sa liste d'adjacence, dans l'autre sens. On utilise directement $\tilde{\delta}$ ce qui permet de travailler implicitement dans $\widetilde{M}$.

OCaml
```
let transpose m =
  let n = Array.length m in
  let n_tilde = n * (n + 1) / 2 in
  let g = Array.make n_tilde [] in
  for q = 0 to n_tilde - 1 do
    for a = 0 to nb_lettres - 1 do
      let r = delta_tilde m q a in
      g.(r) <- (q, char_of_int a) :: g.(r)
    done
  done;
  g
```

Q 22) La question 17 indique que l'existence d'un mot synchronisant est équivalente au fait que dans $\widetilde{M}$ de tout état provenant d'un couple d'état de M on peut rejoindre un état provenant d'un état singleton de M. Il suffit donc de vérifier, dans $\widetilde{G}^T$, que tous les sommets provenant d'un couple d'état de M sont accessibles depuis un sommet provenant d'un état singleton de M. Il suffit donc de lancer la fonction `accessible` avec comme sommets de départs tous ceux provenant d'un état singleton de $\widetilde{M}$ et de vérifier que tous les sommets sont accessibles, c'est-à-dire que le nombre de sommets accessibles est l'ordre du graphe.

Q 23) Il suffit d'appliquer l'idée précédente.

OCaml
```
let existe_synchronisant m =
  let n = Array.length m in
  let g = transpose m in
  let n_tilde = Array.length g in
  let singletons = List.init n (fun i -> phi n [i]) in
  accessibles g singletons = n_tilde
```