Centrale 2017 - Option informatique
Un corrigé

1 Considérations générales

1. Pour une machine a un seul état q_0, tout mot lu depuis l'unique état mène en q_0. Tout mot est synchronisant.
2. Ici l'alphabet est $\Sigma = \{a\}$. Une récurrence simple montre qu'un mot de longueur paire mène de i en i alors qu'un mot de longueur impaire mène de i en $3-i$. Aucun mot ne mène donc simultanément de 1 et 2 dans le même état. Il n'existe donc pas de mot synchronisant.
3. Le mot acb est synchronisant pour M_2.
4. Une fonction récursive locale (et connaissant donc `m`) `lire : mot` $\to$ `etat` $\to$ `etat` donne l'état obtenu par lecture d'un mot à partir d'un état. J'utilise cette fonction locale pour éviter de passer `m` en argument dans les appels récursifs.

```
let delta_etoile m e u  =
   let rec lire u e = match u with
      |[] -> e
      |x::q -> lire q (m.delta e x)
 in lire u e ;;
```

5. On regarde l'état `q0` obtenu par lecture du mot à partir de l'état 0. On teste alors successivement si on atteint le même état à partir de $1, 2, \ldots, p-1$. On peut s'arrêter dès que l'on tombe sur autre chose que `q0` et c'est pourquoi je choisis une boucle conditionnelle.

```
let est_synchronisant m u =
 let q0=delta_etoile m 0 u
 in let i=ref 1
 in while !i<m.n_etats && (delta_etoile m !i u)=q0 do incr i done;
 !i=m.n_etats;;
```

6. Supposons qu'on dispose d'un mot synchronisant u. La machine possède au moins deux états q_0 et q_1 et par définition, on a $q_0.u = q_1.u$.
Comme $q_0.\varepsilon \neq q_1.\varepsilon$, il existe un préfixe v de u tel que $q_0.v \neq q_1.v$. Comme l'ensemble des préfixes de u est fini, il va exister un plus long préfixe v tel que $q_0.v \neq q_1.v$.
Comme $q_0.u = q_1.u$, $v \neq u$ et il existe une lettre x telle que vx soit préfixe de u. Par choix de v, on a $(q_0.v).x = q_0.(vx) = q_1.(vx) = (q_1.v).x$. Les états $q = q_0.v$ et $q' = q_1.v$ et la lettre x conviennent donc.
7. On prouve par récurrence sur la longueur du mot m (ou par induction structurelle) que
$$\forall P \subset Q,\ \widehat{\delta}^*(P, m) = \{\delta^*(p, m),\ p \in P\}$$
   - Pour tout $P \subset Q$, $\widehat{\delta}^*(P, \varepsilon) = P$ et $\{\delta^*(p, \varepsilon),\ p \in P\} = P$. Le résultat est donc vrai pour le mot vide.
   - Supposons le résultat vrai pour un mot m de longueur n. Soit $m' = xm$ un mot de longueur $n+1$. On a
$$\widehat{\delta}^*(P, xm) = \widehat{\delta}^*(\widehat{\delta}(P, x), m)$$
Par hypothèse de récurrence, et en posant $P' = \widehat{\delta}(P, x)$,
$$\widehat{\delta}^*(P, xm) = \{\delta^*(q, m),\ p \in P'\}$$
Or, $P' = \{\delta(p, x)/\ p \in P\}$ et donc
$$\widehat{\delta}^*(P, xm) = \{\delta^*(\delta(p, x), m),\ p \in P\} = \{\delta^*(p, xm),\ p \in P\}$$
ce qui prouve le résultat pour m'.

(a) Supposons qu'il existe un mot u synchronisant pour M et notons q_0 l'état de M où mène la lecture de u. Soit $P = Q$ (considéré comme partie de Q et donc état de $\widehat{M}$). On a alors

$$\widehat{\delta}^*(P, u) = \{\delta^*(p, u),\ p \in P\} = \{q_0\}$$

Réciproquement, supposons qu'il existe un mot u tel que $\widehat{\delta}^*(Q, u)$ soit un singleton $\{q_0\}$. Alors u est synchronisant pour M.
Il existe donc un mot synchronisant pour M si et seulement si il existe dans $\widehat{M}$ un singleton accessible depuis l'état Q.

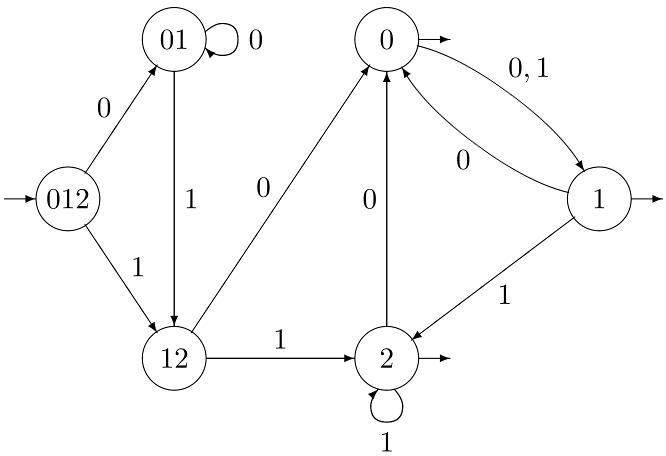
(b) Considérons l'automate obtenu depuis la machine $\widehat{M}$ en prenant Q comme état initial et les singletons comme états terminaux. Les mots reconnus par cet automate sont exactement les éléments de $LS(M)$.

(c) On commence par donner la table de M_0 puis on en déduit celle de $\widehat{M_0}$. On procède comme pour l'algorithme de déterminisation en partant de ce qui sera l'état initial c'est à dire $\{0, 1, 2\}$.

M_0	0	1	2
0	1	0	0
1	1	2	2

$\widehat{M_0}$	$0,1,2$	$0,1$	$1,2$	0	2	1	$0,2$	$\emptyset$
0	$0,1$	$0,1$	0	1	0	0	$0,1$	$\emptyset$
1	$1,2$	$1,2$	2	1	2	2	$1,2$	$\emptyset$

Au delà de la double barre, les états sont non accessibles depuis l'état initial de l'automate reconnaissant $LS(M_0)$ qui est dessiné ci-dessous.



Dès que l'on arrive sur un état singleton, on reste sur ces états. On arrive pour la première fois sur un singleton quand on lit le premier 1 puis un 0 ou un 1. Ainsi

$$LS(M_0) = 0^*1(0+1)(0+1)^*$$

8. Soit M une machine et q_0 un état de M. Notons M' la machine obtenue à partir de M en redirigeant toutes les transitions issues de q_0 vers lui même.
   - Supposons que M' possède un mot synchronisant u. Comme $q_0.u = q_0$, la lecture de u dans M' depuis tout état mène à q_0. On en déduit que la lecture de u depuis tout état dans M passe par q_0.
   - Réciproquement, s'il existe u tel que la lecture de u depuis tout état dans M passe par q_0 alors u est synchronisant pour M' (la lecture de u dans M' depuis tout état mène en q_0).

2 Algorithmes classiques

1. Dans toute la manipulation des files, les indices du tableau devront être considérés modulo la longueur de ce tableau. Quand nous dirons que deux indices sont égaux, cela signifiera toujours "égaux modulo la longueur du tableau".

(a) La première idée est que la file est pleine quand l'indice de début est égal à l'indice de fin. Cela ne me semble pas toujours vrai. Il y a en effet le cas particulier de la file vide pour laquelle aucune convention n'a été choisie pour les indices de début et de fin.
Pour éviter toute ambiguïté et ne pas avoir à choisir une convention et à vérifier sa validité, je préfère traiter à part le cas où la file est vide. Je place alors l'élément en position `f.deb` et je dis que la fin de la file est juste à côté. Sinon, on place l'élément en position `f.fin` et on incrémente ce champ.

```
let ajoute f x =
     if f.vide then begin
           f.vide <- false ;
           f.fin <- (1+f.deb) mod (vect_length f.tab);
           f.tab.(f.deb) <- x
           end
     else if f.fin <> f.deb then begin
           f.tab.(f.fin) <- x ;
           f.fin <- (f.fin + 1) mod (vect_length f.tab)
           end
     else failwith "File pleine";;
```

(b) On stocke l'élément en position `f.deb` (si la file n'est pas vide) et on incrémente ce champ. Il faut bien sûr éventuellement changer la valeur du booléen.

```
let retire f =
     if f.vide then failwith "File vide";
     let x=f.tab.(f.deb) in
     f.deb <- (f.deb + 1) mod (vect_length f.tab);
     f.vide <- ((f.deb - f.fin)= 0) ;
     x;;
```

(c) Ces deux fonctions ont immédiatement une complexité constante $O(1)$.

2. Le seul problème de terminaison vient de la boucle conditionnelle. On a un premier invariant de l'algorithme (qui est immédiat puisque $D[s]$ n'est modifié que lorsque s est ajouté à la file).

$$(I_1) : s \text{ est ou a été dans } F \text{ si et seulement si } D[s] < \infty$$

Or, on n'ajoute un sommet s' à la file que si $D[s'] = \infty$. On en déduit qu'un sommet est ajouté au plus une fois à la file. Comme la boucle **tant que** supprime au moins un élément de la file à chaque étape, cette boucle est effectuée au plus n fois et se termine.

3. Les initialisations ont un coût $O(|S|)$ (créations de tableaux de taille n).
La première boucle est effectuée au plus $|E|$ fois et chaque itération est en temps constant. La première boucle a donc un coût $O(n)$.
Dans la boucle conditionnelle, on regarde au plus une fois la liste des arêtes issues de chaque sommet. Cette boucle a donc un coût $O(|A|)$.
Finalement, le coût de l'algorithme 1 est $O(|S| + |A|)$.

4. On prouve le résultat par récurrence (sur le numéro de la boucle que l'on exécute).
   - Initialement, la file contient les éléments de E et les valeurs de D associées sont toutes nulles. La propriété est ainsi vraie.
   - Supposons la propriété vraie à un instant donné et supposons que la boucle s'effectue. La file contient des sommets $s_1, \dots, s_r$ avec les propriétés voulues pour D. En fin de boucle, la file contiendra des éléments $s_2, \dots, s_r, s'_1, \dots, s'_k$. Les valeurs $D[s_i]$ n'ont pas été modifiées et $D[s'_j] = D[s_1] + 1$. Comme $D[s_r] \leq D[s_1] + 1 = D[s'_1] = \cdots = D[s'_k]$, on a la propriété d'ordre. De plus
   si $r \geq 2$, $D[s'_k] - D[s_2] = D[s_1] + 1 - D[s_2] \leq 1$ et on aussi la seconde propriété
   sinon, $D[s'_k] - D[s'_1] = 0$ et on conclut encore.

5. (a) On a l'invariant suivant :

$$(I_2) : \text{si } D[s] < \infty \text{ alors } s \text{ est accessible depuis un sommet de } E.$$

Supposons, par l'absurde, qu'il existe un sommet accessible depuis un sommet de E et tel que $D[s] = \infty$ en fin d'algorithme. On peut alors choisir parmi ces sommets un élément tel que d_s est minimal. En considérant le prédécesseur t de s dans un chemin de longueur d_s entre un élément de E et s, on a donc $D[t] < +\infty$ (par minimalité) ce qui montre que t est passé par la file (invariant I_1). Mais alors, tous les voisins de t et donc s ont été ajoutés à F et $D[s] = +\infty$, ce qui est une contradiction.
On montre ensuite que l'on a l'invariant

$$(I_3) : \forall s,\ D[s] \text{ est la longueur d'un chemin d'un élément de } E \text{ à } s \text{ quand } D[s] \neq \infty.$$

On en déduit alors a fortiori que $\forall s,\ D[s] \geq d_s$ (immédiat si $D[s] = \infty$ et conséquence de l'invariant sinon).
On décrémente s à chaque fois que l'on ajoute un élément à la file. En fin de boucle, c est le nombre de sommets inaccessibles depuis E.

(b) Montrons que la propriété suivante est un invariant

$$\forall s,\ D[s] \text{ est la longueur d'un chemin minimal d'un élément de } E \text{ à } s \text{ quand } D[s] \neq \infty$$
$$\text{si } s \text{ est en tête de } F, \text{ on a } D[s'] \neq \infty \text{ pour tout sommet } s' \text{ tel que } d_{s'} \leq D[s] = d_s.$$

- La propriété est initialement vraie car on traité correctement tous les éléments de E.
- On suppose que la propriété est vraie et que la boucle s'effectue. On considère l'élément s en tête de F. On modifie alors la valeur $D[s']$ quans s' est un successeur de s. Cette modification ne s'effectue que si $D[s'] = \infty$ et par l'hypothèse de récurrence, $d_{s'} > D[s]$. Comme $D[s'] = D[s]+1$ on a donc $D[s'] \leq d_{s'}$ et avec la question précédente, $d_{s'} = D[s]$. Pour la seconde partie de l'invariant, il n'y a de problème que si la nouvelle tête t de F vérifie $D[t] > D[s]$. Avec la question 4, on a alors $D[t] = D[s] + 1$. Il s'agit donc de voir que tous les sommets à distance $D[s] + 1$ ont été incorporés à F. Ceci est vrai car tous les sommets à distance d_S ont été "traités" (hypothèse de récurrence) puis sortis de F (puisque l'on est dans le cas où $D[t] > D[s]$).

Comme en fin de boucle on a traité tous les sommets accessibles depuis E, on a $d_s = D[s]$ pour tous ces sommets en fin de boucle.
La tableau P permet de reconstruire un chemin minimal entre un sommet de E et un sommet s. $P[s]$ contient en effet ' 'la dernière arête" dans un tel chemin minimal (si $P[s] = (t, y)$, la dernière arête dans ce chemin est (s, t, y)).

6. Si on n'utilise pas de référence de liste pour parcourir tant la liste représentant E que les listes d'adjacence (ce que l'on doit faire pour implémenter les boucles **pour**), il faut écrire des fonctions auxiliaires. On les écrit localement (elle connaissent donc les différents tableaux).
`initier` prend en argument la liste `e` représentant E et modifie les tableaux selon la première boucle **pour** de l'algorithme.
`parcours` prend en argument une liste d'adjacence et le sommet qui lui est associé. Elle modifie les tableaux selon la boucle **pour** enchâssée dans la boucle conditionnelle.

```
let accessibles v e =
 (* cr\'eation des tableaux et de la file *)
   let n=vect_length v in
   let f= {tab=make_vect n 0;deb=0;fin=0;vide=true} in
   let d=make_vect n (-1) in
   let p=make_vect n (-2,-1) in
   let c=ref n in
 (* fonction pour la premi\`ere boucle *)
   let rec initier e = match e with
       |[]->()
```

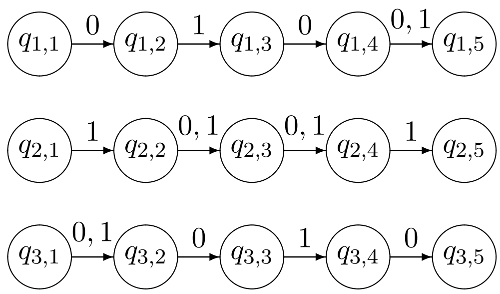
```
        |s::q ->ajoute f s;
                d.(s) <- 0 ;
                p.(s) <- (-1,-1);
                decr c ;
                initier q
  (* fonction de parcours d'une liste d'adjacence pour un sommet *)
in let rec parcours l s= match l with
        |[] -> ()
        |(t,y)::q -> if d.(t)=(-1) then
                 begin
                 d.(t) <- d.(s) + 1;
                 p.(t) <- (s,y);
                 ajoute f t;
                 decr c
                 end ;
                      parcours q s
in initier e;
   while not f.vide do
      let s=retire f in
      parcours v.(s) s
      done;
   (!c,d,p);;
```

7. A partir de s, on trouve le prédécesseur dans un plus court chemin avec $p.(s)$. On obtient (s'il existe) un sommet t à partir duquel on recommence (appel récursif). On notera les points suivants en notant (t, y) la valeur de $p.(s)$.
   - Si $t = -2$, on n'a pas de chemin.
   - Si $t = -1$, $t \in E$ et on a terminé.
   - Le chemin se construit à l'envers (on obtient en premier la dernière lettre). C'est pourquoi on écrit une fonction auxiliaire faisant la construction (`creeliste`). Le résultat à renvoyer est l'image miroir de la liste créée. On aurait pu utiliser un accumulateur pour éviter d'utiliser `rev` (image miroir).

```
let chemin s p =
   let rec creeliste s =
      let (t,y)=p.(s) in
      if y=(-2) then failwith "Chemin inexistant"
      else if y=(-1) then []
      else y::(creeliste t)
 in rev (creeliste s);;
```

3 Reduction SAT

1. On ne représente ici que les transitions qui ne pointent pas vers l'état puits.



2. On peut choisir $(1, 1, 0, 0)$ ou encore $(1, 1, 1, 0)$ et on obtient des mots synchronisants (la lecture à partir de tout état mène à l'état puits).
3. La lecture d'une lettre depuis un état $q_{i,j}$ mène soit dans l'état puits soit dans l'état $q_{i,j+1}$. Un mot de longueur k à partir de $q_{i,j}$ et ne menant pas à f mène donc à $q_{i,j+k}$. Ceci n'est possible que si $j + k \leq m + 1$ et impose $k \leq m$ (puisque $j \geq 1$). Ainsi, un mot de longueur $m + 1$ est synchronisant puisqu'il amène en l'état f.
   Si u est un mot de longueur m. Par le même raisonnement, la lecture de u depuis $q_{i,j}$ avec $j \geq 2$ amène en f. u est donc synchronisant si et seulement si $q_{i,1}.u = f$ pour tout i.
4. On suppose F satisfiable et on considère le mot $v = v_1 \dots v_m$ associé à une distribution de vérité satisfaisant F. D'après la question précédente, on cherche à montrer que $q_{i,1}.v = f$ pour tout i.
   Soit $i \in [|1, n|]$. c_i étant satisfaite par la distribution, il existe un entier j tel que ($v_j = 1$ et x_j est présent dans c_i) ou ($v_j = 0$ et $\overline{x_j}$ est présent dans c_i) car une disjonction est satisfaite quand un des littéraux qui la compose l'est. On peut alors considérer le premier tel j. On a $\delta(q_{i,1}, v_1 \dots v_{j-1}) = q_{i,j}$ et la lecture de v_j mène en f.
   Ceci montre que v est un mot synchronisant.
5. Supposons, que l'on dispose d'un mot synchronisant $v = v_1 \dots v_k$ avec $k \leq m$. Comme f est un état puits, c'est l'état commun où l'on aboutit par lecture de v depuis tout état. Ainsi $\forall i,\ q_{i,1}.v = f$. Considérons la distribution donnant la valeur v_i à x_i si $i \leq k$ et une valeur quelconque arbitraire si $i \geq k + 1$.
   Supposons, par l'absurde, que la clause c_i ne soit pas satisfaite par cette distribution. On montre alors comme en question précédente que $\delta^*(q_{i,1}, v_1 \dots v_k) = q_{i,k+1}$ ce qui est contradictoire. Toutes les clauses sont donc satisfaites et F est satisfiable (et on a trouvé une distribution associée).

4 Existence

1. (a) Comme les machines considérées sont déterministes, la lecture d'une lettre depuis un ensemble d'états fait diminuer (au sens large) le nombre d'états. La suite de terme général $|Q.u_i|$ est donc décroissante. De plus, $|Q.u_r| = 1$ car le mot u est synchronisant.
   (b) S'il existe un mot synchronisant u alors $\forall q, q',\ q.u = q'.u$ et on peut choisir $u_{q,q'} = u$ pour satisfaire la propriété.
   Supposons maintenant que pour chaque choix de q et q' on ait l'existence d'un mot $u_{q,q'}$. La machine possède au moins deux états q_0, q_1 et on peut trouver un mot m_0 tel que $q_0.m_0 = q_1.m_0$. Ainsi, $|Q.m_0|$ est de cardinal au plus $n-1$. Si $|Q.m_0| = 1$, on s'arrête. Sinon, $Q.m_0$ possède au moins deux états et on trouve un mot m_1 tel que $|(Q.m_0).m_1| < |Q.m_0|$. On a donc $|Q.(m_0m_1)| \leq n - 2$. On va alors pouvoir poursuivre la construction. On peut formaliser à l'aide d'une récurrence en construisant pour tout $k \in [|0, n - 2|]$ un mot u_k tel que $|Q.u_k| \leq n - k - 1$. Le mot u_{n-2} est alors synchronisant.

2. Il y a n parties à 1 élément et $\binom{n}{2} = \frac{n(n-1)}{2}$ parties à 2 éléments. Ainsi

$$\widetilde{n} = n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2}$$

3. On calcule l'état q associé à $\overline{q}$. C'est une liste qui a un ou deux éléments qui sont des états de `m`. On regarde vers quels états nous mènent ces éléments (ou cet élément). On obtient un ou deux états de `m` que l'on transforme en une liste (ordonnée) puis en un entier qui est l'état cherché de $\widetilde{M}$.

```
let delta2 m e x = match (nb_to_set m.n_etats e) with
    |[i] -> set_to_nb m.n_etats [m.delta i x]
    |[i;j] -> let ei=m.delta i x
              and ej=m.delta j x
              in if ei=ej then set_to_nb m.n_etats [ei]
                 else if ei<ej then set_to_nb m.n_etats [ei;ej]
                 else set_to_nb m.n_etats [ej;ei];;
```

4. On crée un tableau `vr` de bonne taille $\widetilde{n}$ composé de listes vides. Pour chaque état $\overline{q}$ de $\widetilde{M}$ et chaque lettre x, `delta2` permet d'obtenir l'état atteint dans $\widetilde{M}$. On a donc une transition $(\overline{q}, x, \overline{q}')$ de $\widetilde{M}$. Ceci signifie que dans $\widetilde{G}_R$, on a un arc de $\overline{q}'$ vers $\overline{q}$ d'étiquette x et on met donc la case $\overline{q}'$ de `vr` à jour.

```
let retourne_machine m =
  let n=m.n_etats in
  let ntilde=n*(n+1)/2 in
  let vr=make_vect ntilde [] in
  for e=0 to ntilde-1 do
     for x=0 to m.n_lettres-1 do
        let f=delta2 m e x in
        vr.(f) <- (e,x)::vr.(f)
        done ;
     done;
  vr;;
```

5. D'après la question **4.1b**, il existe un mot synchronisant si depuis tout état du type $[i;j]$ $(i < j)$ de $\widetilde{M}$ il existe un mot $u_{i,j}$ qui nous amène dans un état $[k]$. Comme la lecture d'un mot dans $\widetilde{M}$ depuis un état $[i]$ nous amène toujours dans un état $[k]$, ceci revient à voir si tout état de $\widetilde{M}$ permet d'atteindre un état du type $[k]$.
Il s'agit donc de voir si dans $\widetilde{G}$ il existe un chemin de tout état vers un état singleton. Ceci revient à voir si dans $\widetilde{G}_R$ on peut atteindre tout état depuis les états singleton.
Il suffit donc d'appliquer `accessibles` depuis l'ensemble des états singleton à $\widetilde{G}_R$ et de voir si on peut atteindre tous les sommets du graphe.

6. La fonction `construit : int` $\rightarrow$ `int` $\rightarrow$ `int list` est telle que `construit n k` crée la liste des états $[i]$ de $\widetilde{M}$ avec $i = k, \dots, n-1$. On utilise alors `accessibles` avec cette liste et $\widetilde{G}_R$ comme indiqué en question précédente. On obtient un triplet dont le premier élément donne le nombre d'éléments non accessibles et on regarde s'il est nul.

```
let rec construit n k =
   if k=n then []
   else (set_to_nb n [k])::(construit n (k+1));;

let existe_synchronisant m =
   let (c,d,p)=accessibles (retourne_machine m) (construit m.n_etats 0)
   in c=0;;
```