I Considérations générales

1. Si M a un unique état q_9, alors tout mot, lu depuis l'unique état q_0, mène en q_0 : tout mot est synchronisant.

2. Soit $w \in \Sigma^*$.
   - Si w est de la forme a^{2n}, alors une récurrence immédiate montre que $q_1.w = q_1$ et $q_2.w = q_2$, donc w n'est pas synchronisant.
   - Si w est de la forme a^{2n+1}, on a $q_1.w = q_2$ et $q_2.w = q_1$, donc w n'est pas synchronisant.

   Donc M_1 n'admet pas de mot synchronisant.

3. *aca* convient, on aboutit nécessairement en q_2.

4.
```
let rec delta_etoile m q w =
  match w with
  | [] -> q
  | x :: xs -> delta_etoile m (m.delta q x) xs
```

5. On peut faire une boucle `while`, une fonction auxiliaire récursive ou une boucle `for` dont on sort *via* une exception. Une boucle `for` parcourue systématiquement jusqu'au bout n'était visiblement pas du goût du jury : « *Même si ce n'est pas explicitement indiqué, quand on teste si un mot est, ou non, synchronisant, il semble normal de s'arrêter dès qu'on vérifie qu'il ne l'est pas ; beaucoup de candidats calculent tout, et même parfois deux fois, avant de conclure.* »

```
let est_synchronisant m w =
  let candidat = delta_etoile m 0 w in
  let q = ref 1 in
  while !q < m.n_etats && delta_etoile m !q w = candidat do
    incr q
  done;
  !q = m.n_etats
```

6. Supposons que M (à au moins deux états) possède un mot synchronisant $u = u_1 \dots u_n$ et notons $u_{\leq i} = u_1 \dots u_i$ le préfixe de longueur i de u. Soient alors $q_1 \neq q_2$ tels que $q_1.u = q_2.u$. L'ensemble $X = \{i \in [1 \dots n] \mid q_1.u_{\leq i} = q_2.u_{\leq i}\}$ est non vide (il contient n) et minoré par 1 : notons alors j son plus petit élément. En posant $q = q_1.u_{\leq j-1}$ et $q' = q_2.u_{\leq j-1}$ (avec $u_{\leq j-1} = \varepsilon$ si $j = 1$), on a :
   - $q.u_j = q.u_{\leq j} = q'.u_{\leq j} = q'.u_j$ puisque $j \in X$;
   - $q \neq q'$ puisque $j - 1 \notin X$.

   Les états q et q' conviennent donc.

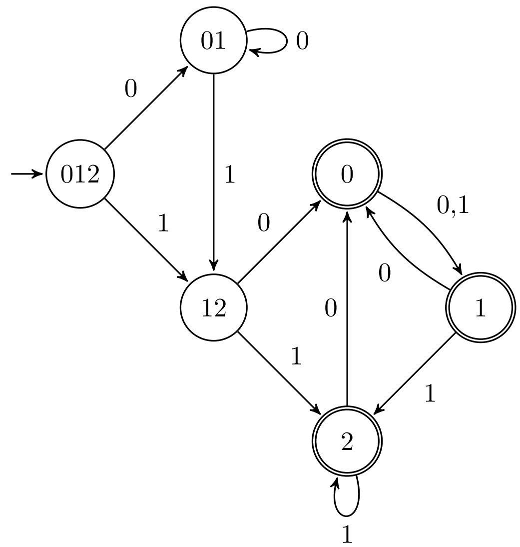
7. Une récurrence sur la longueur du mot u montre que $\widehat{\delta}^*(P, x) = \{\delta^*(q, x) \mid q \in P\}$ pour toute partie P de Q. On a donc :

$$\begin{aligned} u \text{ synchronisant} &\Longleftrightarrow \exists q_0 \in Q,\ \forall q \in Q,\ \delta^*(q, u) = q_0 \\ &\Longleftrightarrow \exists q_0 \in Q,\ \widehat{\delta}^*(Q, u) = \{q_0\} \end{aligned}$$

   Ainsi, la machine M admet un mot synchronisant si et seulement si il existe un état singleton $\{q_9\}$ accessible depuis l'état Q dans la machine des parties $\widehat{M}$.

8. Transformons la machine $\widehat{M}$ en automate en prenant Q comme état initial et les singletons comme états acceptants. Les mots synchronisants pour M sont exactement les mots acceptés par cet automate.

9. On ne construit que les états accessibles depuis l'état initial $\{0, 1, 2\}$:



On observe que, dès que l'on arrive sur un état acceptant, on boucle forcément sur des états acceptants. On arrive dans l'état $\{1, 2\}$ en lisant $0^+1|1 \equiv 0^*1$, puis sur un état singleton en lisant $0|1$. On obtient donc :

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

10. Soient $M = (\Sigma, Q, \delta)$ et $q_0 \in Q$, et considérons la machine $M' = (\Sigma, Q, \eta)$ où :

    - $\eta(q_0, a) = q_0$ pour tout $a \in \Sigma$;
    - $\eta(q, a) = \delta(q, a)$ si $q \neq q_0$.

    Il est immédiat que pour tout mot u et état q, on a $\eta^*(q, u) = q_0$ si et seulement si la lecture de u à partir de q dans M fait passer par q_0.

    - S'il existe u synchronisant dans M', alors la synchronisation se fait nécessairement sur l'état q_0 puisque $\eta^*(q_0, u) = q_0$. Donc pour tout état q on a $\eta^*(q, u) = q_0$ et la lecture de u depuis q dans M fait passer par q_0.
    - Inversement, s'il existe u tel que lire u depuis q dans M fait nécessairement passer par q_0, alors lire u depuis q dans M' fait nécessairement *aboutir* en q_0 : u est synchronisant dans M'.

II Algorithmes classiques

11. Si l'on a $deb = fin$ (égal 0, par exemple), on peut être dans deux situations différentes :

    - soit une file vide ;
    - soit une file « pleine ».

    Par exemple, si l'on a une file à un élément avec $deb = 0$ et $fin = 1$ et qu'on retire cet élément, on aura une file vide avec $deb = fin = 1$. D'un autre côté, avec un tableau de taille n, si l'on a $deb = 0$ et $fin = n - 1$ (file contenant $n - 1$ éléments) et que l'on ajoute un élément, on aura $deb = fin = 0$ et la file aura n éléménts. Pour distinguer entre les deux possibilités, le champ `vide` est nécessaire (ou quelque chose d'autre, comme par exemple un champ `cardinal`).

12. Comme vu au-dessus, la file est pleine si $deb = fin$ et qu'elle n'est pas vide. On n'oublie pas de mettre à jour `f.vide` après l'ajout :

```
let ajoute f x =
  if f.deb = f.fin && not f.vide then failwith "file pleine";
  f.tab.(f.fin) <- x;
  f.fin <- (f.fin + 1) mod (Array.length f.tab);
  f.vide <- false
```

13. Si $deb = fin$ juste après une extraction, alors on vient de vider la file.

```ocaml
let retire f =
  if f.vide then failwith "file vide";
  let resultat = f.tab.(f.deb) in
  f.deb <- (f.deb + 1) mod (Array.length f.tab);
  if f.deb = f.fin then f.vide <- true;
  resultat
```

14. Les deux fonctions ont une complexité en $\mathrm{O}(1)$ (immédiat).

15. La seule question concerne la boucle **`while`**. On remarque que pour tout sommet s :

    - $D[s] = \infty$ si et seulement si s n'a jamais été ajouté à la file ;
    - si $D[s] \neq \infty$, alors $D[s]$ ne sera plus modifié et s ne sera plus jamais ajouté à la file.

    Un sommet est donc ajouté au plus une fois à la file, or on enlève un sommet de la file à chaque tour de la boucle **`while`**. Il y a donc au plus n tours, ce qui garantit la terminaison.

16. - L'initialisation de F, D et P se fait en temps $\mathrm{O}(|S|)$.
    - On passe $|X| \leq |S|$ fois dans la première boucle, dont le corps est en $\mathrm{O}(1)$, donc $\mathrm{O}(|S|)$ au total pour cette boucle.
    - Ensuite, un sommet est retiré de la file au plus une fois, et on parcourt alors la liste d'arcs associée, en faisant des opérations en temps constant sur chaque arc. La boucle **`while`** se fait donc en $\mathrm{O}(|A| + |S|)$.

    Au total, la complexité est en $\mathrm{O}(|S| + |A|)$.

17. Au début de la première itération, c'est vrai : les seuls sommets dans F sont ceux de X, et $D[s] = 0$ pour ces sommets.

18. Supposons l'invariant vérifié au début d'une itération, et F non vide (sinon on sort de la boucle). On extrait s_1 et l'on insère en fin de file ses successeurs $s'_1, \ldots, s'_p$ non encore vus, en fixant $D[s'_i] = D[s_1] + 1$. On a donc en fin d'itération $s_2, \ldots, s_r, s'_1, \ldots, s'_p$ avec
$$\underbrace{s_2 \leq \cdots \leq s_r \leq D[s_1] + 1}_{\text{d'après l'invariant}} = D[s'_1] = \cdots = D[s'_p]$$
    De plus, $D[s'_p] - D[s_2] \leq D[s'_p] - D[s_1] = 1$ (ou alors $r = 1$ et la file est $s'_1, \ldots, s'_p$) donc l'invariant est conservé.

19. On a facilement l'invariant suivant, valable à tout moment de l'exécution : « si $D[s] < \infty$, alors s est accessible depuis un sommet de X ». On en déduit un sens de l'équivalence.

    Pour l'autre sens, supposons que s soit accessible depuis un sommet q de X et soit $q = s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_p = s$ un chemin de q à s. En fin d'exécution, on a :

    - $D[s_0] = 0$, fixé dans la première boucle **`for`** ;
    - pour $0 \leq i < p$, si $D[s_i] < \infty$, alors $D[s_{i+1}] < \infty$. En effet, quand on a ajouté s_i à F, on a considéré tous ses successeurs, dont s_{i+1} : soit la valeur de $D[s_{i+1}]$ était déjà autre que ∞, soit elle a été modifiée à ce moment.

    Par récurrence finie, on en déduit $D[s_p] = D[s] < \infty$ en fin d'exécution, ce qui prouve l'équivalence demandée.
    c commence à n et est décrémenté à chaque fois qu'une valeur de $D[s]$ est modifiée : en fin d'éxécution, c donne le nombre de sommets inaccessibles depuis X.

20. On a l'invariant suivant, valable au début de chaque itération de la boucle **`while`** :

    - si $D[s] < \infty$, alors $D[s] = d_s$;
    - si $D[s] = \infty$ et si la file est non vide, alors $d(s) > d_{s_1}$ (où s_1 est l'élément en tête de file).

    Initialisation. Au début de la première itération, les seuls éléments pour lesquels $D[s] < \infty$ sont ceux de X. On a bien $D[s] = d_s = 0$ pour ces éléments. Tous les autres éléments vérifient $d_s > 0$, donc le deuxième point est également correct.

    Invariance. Il y a deux points à vérifier :

    - pour les éléments que l'on rajoute à F, il faut avoir $D[s] = d_s$. On fixe $D[s]$ à $D[s_1] + 1$ pour ces éléments, et l'invariant fournit $d_s \geq d_{s_1} + 1 = D[s_1] + 1$. Pour l'autre inégalité, on remarque que ces s sont des successeurs de s_1, et vérifient donc $d_s \leq d_{s_1} + 1$.

- pour les éléments qui ne sont toujours pas dans la file, il faut vérifier $d_s > d_{s_2}$ (si s_2 existe). Considérons un éventuel plus court chemin $q \in X \longrightarrow \ldots \longrightarrow s' \longrightarrow s$, et remarquons qu'on a nécessairement $d_s = d_{s'} + 1$. Si s' était sorti de la file, alors s y aurait été ajouté. Donc :
  - soit $D[s'] = \infty$, donc $d_{s'} > d_{s_1}$ et $d_s = d_{s'} + 1 > d_{s_1} + 1$. D'après une question précédente, cela garantit $d_s > d_{s_2}$.
  - soit $D[s'] < \infty$, et donc s' est dans la file. Toujours d'après une question précédente, on a alors $D[s'] \geq D[s_2]$, et d'un autre côté notre invariant garantit $d_{s'} = D[s']$ et $d_{s_2} = D[s_2]$. À nouveau, on a $d_s = d_{s'} + 1 > d_{s_2}$.

Conclusion. En fin d'exécution, on a pour tout sommet s :

- soit $D[s] = \infty$, donc s inaccessible d'après la question précédente et $d_s = \infty$;
- soit $D[s] < \infty$, et donc $d_s = D[s]$ d'après notre invariant.

Dans tous les cas, $d_s = D[s]$ en fin d'exécution. $P[s]$ fournit alors couple (q, x) tel que l'arc (q, x, s) soit le dernier arc d'un plus court chemin de X à s.

21.

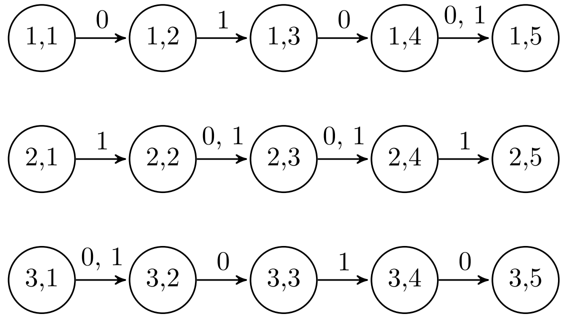
```
let accessibles graphe ens =
  (* initialisation *)
  let n = Array.length graphe in
  let f = {tab = Array.make n 0; deb = 0; fin = 0; vide = true} in
  let inf = -1 in
  let d = Array.make n inf in
  let p = Array.make n Vide in
  let c = ref n in

  (* boucle for *)
  let initialise s =
    ajoute f s;
    d.(s) <- 0;
    p.(s) <- Rien;
    c := !c - 1 in
  List.iter initialise ens;

  (* boucle while *)
  while not f.vide do
    let s = retire f in
    let traite_arc (s', y) =
      if d.(s') = inf then begin
        d.(s') <- d.(s) + 1;
        p.(s') <- Arc (s, y);
        ajoute f s';
        decr c
      end in
    List.iter traite_arc graphe.(s);
  done;

  !c, d, p
```

22. Pour obtenir le chemin dans le bon ordre, il faut utiliser une fonction auxiliaire ou renverser la liste avant de la renvoyer.

```
let chemin s p =
  let rec aux s chemin =
  match p.(s) with
  | Vide -> failwith "pas de chemin"
  | Rien -> chemin
  | Arc (s', x) -> aux s' (x :: chemin) in
  aux s []
```

III Réduction SAT

23. On ne représente pas l'état puits.



24. On vérifie facilement que $(1, 1, 1, 0)$ satisfait F_1, et que le mot 1110 mène à f depuis tout état, et est donc synchronisant.

25. Les transitions ne menant pas à f vont d'un état $q_{i,j}$ à l'état $q_{i,j+1}$. Par conséquent, une suite de $m+1$ transitions :

    - soit passe par f, et y reste donc (c'est un puits) ;
    - soit fait passer de $q_{i,j}$ à $q_{i,j+m+1}$. Mais c'est impossible puisque cela impliquerait $j + m + 1 \leq m + 1$ et donc $j \leq 0$.

    Ainsi, tout mot de longueur $m+1$ est synchronisant (sur l'état puits). Si u est de longueur m, le même raisonnement montre que $q_{i,j}.u = f$ pour tout $j \geq 2$. Donc u est synchronisant si et seulement si les $q_{i,1}.u$ sont tous égaux à f.

26. Soit $(u_1, \dots, u_m)$ une valuation satisfaisant F. Pour toute clause c_i, notons j le numéro du premier littéral de c_i satisfait par u (tel que x_j présent dans c_i et $u_j = 1$, ou $\overline{x_j}$ présent dans c_i et $u_j = 0$). Un tel littéral existe forcément puisque la clause est satisfaite par u. On a alors $q_{i,1}.u_1 \dots u_{j-1} = q_{i,j}$ et donc $q_{i,1}.u_1 \dots u_j = q_{i,j}.u_j = f$. On reste ensuite dans l'état puits, et l'on a donc $q_{i,1}.u = f$. D'après la question précédente, on en déduit que u est synchronisant.

27. Inversement, soit v un mot synchronisant de longueur $k \leq m$. Notons qu'on synchronise nécessairement sur f puisque c'est un état puits. On étend v en une valuation u en posant $u_j = 0$ si $k < j \leq m$. Pour tout i, on a $q_{i,1}.v = f$, ce qui signifie qu'il existe un $j \leq k$ tel que $q_{i,j}.v_j = f$, et donc $q_{i,j}.u_j = f$. On en déduit que u satisfait la clause c_i. Ceci étant le cas pour toutes les clauses, u satisfait F.

IV Existence

28. Comme la machine est déterministe, on a $|E.x| \leq |E|$ pour tout E et pour tout $x \in \Sigma$. On en déduit que la suite $|Q.u_i|$ est décroissante (au sens large). De plus, comme u est synchronisant, on a $|Q.u_r| = 1$.

29. Si u est synchronisant, alors on peut prendre $u_{q,q'} = u$ pour tout couple (q, q').
    Inversement, supposons l'existence des $u_{q,q'}$ et construisons un mot u synchronisant.

    On note $n = |Q|$. Nous allons construire $u = v_1 \dots v_{n-1}$ (les v_i sont des mots, pas des lettres) de manière incrémentale en maintenant l'invariant suivant : $|Q.v_1 \dots v_k| \leq n - k$.

    - Pour $k = 0$, on a bien $|Q.\varepsilon| = n$.
    - Pour $0 \leq k \leq n - 2$:
        - Si $|Q.v_1 \dots v_k| = 1$, on pose $v_{k+1} = \varepsilon$ (on s'arrête, essentiellement).
        - Sinon, on choisit $q, q' \in |Q.v_1 \dots v_k|$, $q \neq q'$ et l'on pose $v_{k+1} = u_{q,q'}$. On a donc $|Q.v_1 \dots v_{k+1}| < |Q.v_1 \dots v_k|$.

      Dans les deux cas, on a $|Q.v_1 \dots v_{k+1}| \leq n - k - 1$.

    Ainsi, $|Q.u| = |Q.v_1 \dots v_{n-1}| \leq 1$, on a en fait égalité et u est synchronisant.

30. On prend les parties à un ou deux éléments d'un ensemble à n éléments, donc $\widetilde{n} = \binom{n}{2} + \binom{n}{1} = \dfrac{n(n+1)}{2}$.

31. Un singleton donne toujours un singleton, mais une paire peut donner une paire ou un singleton :

```
let delta2 m e x =
  let n = m.n_etats in
  match nb_to_set n e with
  | Un i -> set_to_nb n (Un (m.delta i x))
  | Deux (i, j) ->
    let ei = m.delta i x in
    let ej = m.delta j x in
    if ei = ej then set_to_nb n (Un ei)
    else set_to_nb n (Deux (min ei ej, max ei ej))
```

32. Il faut faire attention à la taille du tableau, mais en dehors de cela c'est une question on ne peut plus classique de construction du graphe miroir.

```
let retourne_machine m =
  let n = m.n_etats in
  let ntilde = n * (n + 1) / 2 in
  let v = Array.make ntilde [] in
  for q = 0 to ntilde - 1 do
    for x = 0 to m.n_lettres - 1 do
      let q' = delta2 m q x in
      v.(q') <- (q, x) :: v.(q)
    done
  done;
  v
```

33. D'après la question 28, il existe un mot synchronisant si et seulement si, pour tout couple d'état (q, q') avec $q \neq q'$, on peut trouver un mot $u_{q,q'}$ tel que $q.u_{q,q'} = q'.u_{q,q'}$. Cela revient à dire qu'il y a un singleton accessible depuis l'état $\{q, q'\}$ dans le graphe $\widetilde{G}$. Mais cette condition équivaut au fait que $\{q, q'\}$ est accessible depuis au moins un singleton dans le graphe $\widetilde{G}_R$.

    On peut donc faire dans $\widetilde{G}_R$ un parcours depuis l'ensemble des singletons : si l'on atteint tous les états, la condition de la question 28 est vérifiée et il y a un mot synchronisant, sinon il y a une paire $\{q, q'\}$ inaccessible (les singletons sont évidemment accessibles), la condition n'est pas vérifiée et il n'y a pas de mot synchronisant. et donc la condition des sommets accessibles depuis

34. C'est immédiat avec la question précédente :

```
let singletons n =
  let rec aux k =
    if k = n then []
    else set_to_nb n (Un k) :: aux (k + 1) in
  aux 0

let existe_synchronisant m =
  let c, _, _ = accessibles (retourne_machine m) (singletons m.n_etats) in
  c = 0
```