DS 2-B

# Centrale 2017 MP, mots synchronisants

MPI/MPI*, lycée Faidherbe

**Notations**

- On appelle machine tout triplet $(Q, \Sigma, \delta)$ où $Q$ est un ensemble fini non vide dont les éléments sont appelés *états*, $\Sigma$ un ensemble fini non vide appelé *alphabet* dont les éléments sont appelés *lettres* et $\delta$ une application de $Q \times \Sigma$ dans $Q$ appelée *fonction de transition*. Une machine correspond donc à un automate déterministe complet sans notion d'état initial ou d'états finaux.
- Pour un état $q$ et une lettre $x$, on note $q.x = \delta(q, x)$.
- L'ensemble des mots (c'est-à-dire des concaténations de lettres) sur l'alphabet $\Sigma$ est noté $\Sigma^*$.
- Le mot vide est noté $\varepsilon$.
- On note $ux$ le mot obtenu par la concaténation du mot $u$ et de la lettre $x$.
- On note $\delta^*$ l'extension à $Q \times \Sigma^*$ de la fonction de transition $\delta$ définie par

$$\begin{aligned} \forall q \in Q \quad & \delta^*(q, \varepsilon) = q \\ \forall (q, x, u) \in Q \times \Sigma \times \Sigma* \quad & \delta * (q, xu) = \delta^*(\delta(q, x), u) \end{aligned}$$

- Pour un état $q$ de $Q$ et un mot $m$ de $\Sigma^*$, on note encore $q.m$ pour désigner $\delta * (q, m)$.

Pour deux états $q$ et $q'$, $q'$ est dit accessible depuis $q$ s'il existe un mot $u$ tel que $q' = q.u$.
On dit qu'un mot $m$ de $\Sigma^*$ est *synchronisant* pour une machine $(Q, \Sigma, \delta)$ s'il existe un état $q$ de $Q$ tel que pour tout état $q$ de $Q$, $q.m = q_0$.
L'existence de tels mots dans certaines machines est utile car elle permet de ramener une machine dans un état particulier connu en lisant un mot donné (donc en pratique de la "réinitialiser" par une succession précise d'ordres passés à la machine réelle).

- La partie 1 étudie quelques considérations générales sur les mots synchronisants,
- la partie 2 est consacrée à des problèmes algorithmiques classiques,
- la partie 3 relie le problème de la satisfiabilité d'une formule logique à celui de la recherche d'un mot synchronisant de longueur donnée dans une certaine machine,
- la partie 4 étudie l'existence d'un mot synchronisant pour une machine donnée.

Les parties 1,2 et 3 peuvent être traitées indépendamment.
La partie 4, plus technique, utilise la partie 2.
Dans les exemples concrets de machines donnés plus loin, l'ensemble d'états peut être quelconque, de même que l'alphabet : $\Sigma = \{0, 1\}, \{a, b, c\}, \ldots$.
Par contre, pour la modélisation en OCaml, l'alphabet $\Sigma$ sera toujours considéré comme étant un intervalle d'entiers $[|0, p-1|]$ où $p = |\Sigma|$. Une lettre correspondra donc à un entier entre 0 et $p-1$.
Un mot de $\Sigma^*$ sera représenté par une liste de lettres (donc d'entiers). De même, l'ensemble d'états $Q$ d'une machine sera toujours considéré comme étant l'intervalle d'entiers $[|0, n-1|]$ où $n = |Q|$.

```
type lettre = int
type mot = lettre list
type etat = int
```

Ainsi, la fonction de transition $\delta$ d'une machine sera modélisée par une fonction OCaml de signature `etat -> lettre -> etat`. On introduit alors le type machine

```
type machine = {n_etats : int ;
                n_lettres : int ;
                delta : etat -> lettre -> etat}
```

`n_etats` correspond au cardinal de $Q$, `n_lettres` à celui de $\Sigma$ et `delta` à la fonction de transition. Pour une machine nommée `M`, les syntaxes `M.n_etats`, `M.n_lettres` ou `M.delta` permettent d'accéder à ses différents paramètres. Dans le problème, on suppose que `M.delta` s'exécute toujours en temps constant
Par exemple, on peut créer une machine `m0` à trois états sur un alphabet à deux lettres ayant comme fonction de transition la fonction `f0` donnée ci-après.

```
let f0 etat lettre =
   match etat,lettre with
   |0,0 -> 1
   |0,1 -> 1
   |1,0 -> 0
   |1,1 -> 2
   |2,0 -> 0
   |2,1 -> 2

let m0={n_etats=3;n_lettres=2;delta=f0}
```

La figure 1 fournit une représentation de la machine $M_0$ On pourra observer que les mots 11 et 10

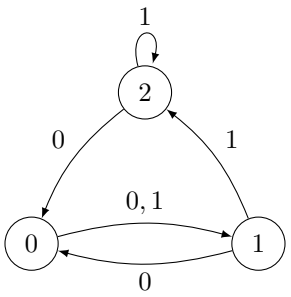



FIGURE 1 – La machine $M_0$

sont tous les deux synchronisants pour la machine $M_0$.
Dans tout le sujet, si une question demande la complexité d'un programme ou d'un algorithme, on attend une complexité temporelle exprimée en $\mathcal{O}(\dots)$.

# I Considérations générales

**Question 1** Que dire des mots synchronisants pour une machine ayant un seul état ?

*Dans une machine à un seul état tout mot envoie cet état dans lui-même donc est synchronisant.*

Dans toute la suite du problème, on supposera que les machines ont au moins deux états.
On considère la machine $M_1$ représentée figure 2.

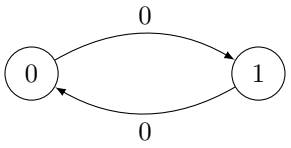



FIGURE 2 – La machine $M_1$

**Question 2** Donne un mot synchronisant pour $M_1$ s'il en existe un. Justifier la réponse.

*Dans $M_1$ on a $\delta^*(1, a^n) = 1 \neq 2 = \delta^*(2, a^n)$ si $n$ est pair et $\delta^*(1, a^n) = 2 \neq 1 = \delta^*(2, a^n)$ si $n$ est impair donc aucun mot sur $\{a\}$ ne synchronise $M_1$.*

On considère la machine $M_2$ représentée figure 3.

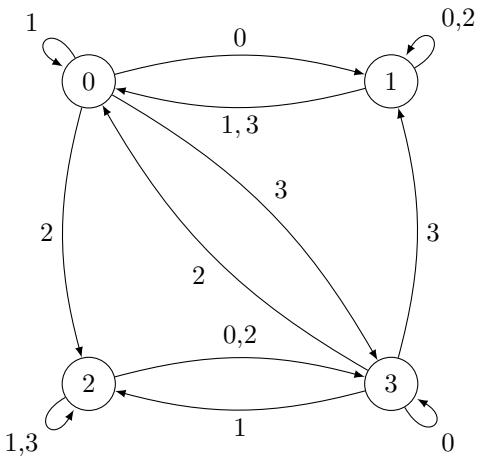



FIGURE 3 – $M_2$ : une machine à 4 états

**Question 3** Donner un mot synchronisant de trois lettres pour $M_2$. On ne demande pas de justifier sa réponse.

*ada, entre autres, est synchronisant.*

**Question 4** Écrire une fonction `delta_etoile : machine -> etat -> mot -> etat` qui, prenant en entrée une machine $M$, un état $q$ et un mot $u$, renvoie l'état atteint par la machine $M$ en partant de l'état $q$ et en lisant le mot $u$.

```
let rec delta_etoile mch etat mot =
  match mot with
  |[] -> etat
  |x::u -> delta_etoile mch (mch.delta etat x) u;;
```

**Question 5** Écrire une fonction `est_synchronisant : machine -> mot -> bool` qui, prenant en entrée une machine $M$ et un mot $u$, dit si le mot est synchronisant pour $M$.

```
let est_synchronisant mch mot =
  let n = mch.n_etats in
  let sync = ref true in
  let fin = delta_etoile mch 0 mot in
  let i = ref 1 in
  while !sync && !i < n do
    sync := (delta_etoile mch !i mot = fin) && !sync;
    i := !i + 1 done;
  !sync;;
```

**Question 6** Montrer que pour qu'une machine ait un mot synchronisant, il faut qu'il existe une lettre $x$ et deux états distincts de $Q$, $q$ et $q'$, tels que $q.x = q'.x$.

*Pour tout mot $u$ on note $\varphi_u$ la fonction de $Q$ dans $Q$ : $q \mapsto \delta^*(q,u)$.*
*Si $u = x_1x_2\cdots x_p$ alors $f_u = f_{x_p} \circ \cdots \circ f_{x_1}$.*
*L'existence d'un mot synchronisant implique l'existence d'un mot $u$ tel que $f_u$ n'est pas surjective car son image est réduite à un état et $Q$ contient au moins deux état. Parmi les lettres de $u$ il en existe au moins une, $y$, telle que $f_y$ n'est pas bijective car sinon $f_u$ le serait.*
*Comme $Q$ est fini, $f_y$ non bijective ne peut être injective donc*
*il existe $q \neq q'$ tel que $f_y(q) = \delta(q,y) \neq f_y(q') = \delta(q',y)$.*

Soit $LS(M)$ le langage des mots synchronisants d'une machine $M = (Q, \Sigma, \delta)$.
On introduit la machine des parties $\widehat{M} = (\widehat{Q}, \Sigma, \widehat{\delta})$ où $\widehat{Q}$ est l'ensemble des parties de $Q$ et où $\widehat{\delta}$ est définie par $\forall P \subset Q,\ \forall x \in \Sigma,\ \widehat{\delta}(P,x) = \{\delta(p,x),\ p \in P\}$.

**Question 7** Justifier que l'existence d'un mot synchronisant pour $M$ se ramène à un problème d'accessibilité de certain(s) état(s) depuis certain(s) état(s) dans la machine des parties.

*$u$ est synchronisant si et seulement si $\hat{\delta}^*(Q,x)$ est un singleton. $M$ admet donc un mot synchronisant si et seulement si un singleton est accessible depuis l'état $Q$.*

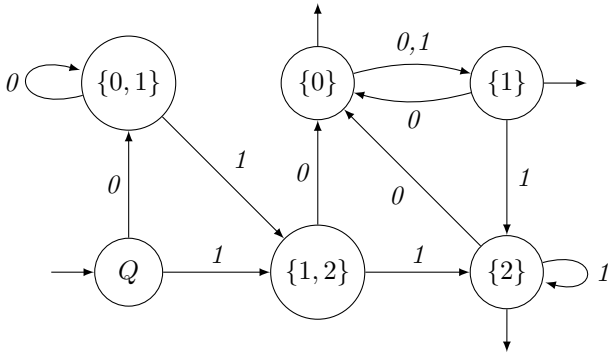
**Question 8** En déduire que le langage $LS(M)$ des mot synchronisants de la machine $M$ est reconnaissable.

*$LS(M)$ est donc reconnaissable car c'est l'ensemble des mot reconnus par l'automate $(\widehat{Q}, \Sigma, \hat{\delta}, Q, Q_1)$ où $Q_1$ est l'ensemble des singletons de $Q$.*

**Question 9** Déterminer la machine des parties associée à la machine $M_0$ puis donner une expression régulière du langage $LS(M_0)$.

*L'automate des parties utile depuis* $\{0,1,2\}$ *s'écrit*

| | $\{0,1,2\}$ | $\{0,1\}$ | $\{1,2\}$ | $\{0\}$ | $\{1\}$ | $\{2\}$ |
|---|---|---|---|---|---|---|
| *0* | $\{0,1\}$ | $\{0,1\}$ | $\{0\}$ | $\{1\}$ | $\{0\}$ | $\{0\}$ |
| *1* | $\{1,2\}$ | $\{1,2\}$ | $\{2\}$ | $\{1\}$ | $\{2\}$ | $\{2\}$ |




*Les mots reconnus forment donc le langage dénoté par* $(0 \cdot 0 * \cdot 1 | 1) \cdot (0|1) \cdot (0|1)*$.

**Question 10** Montrer que si l'on sait résoudre le problème de l'existence d'un mot synchronisant, on sait dire, pour une machine $M$ et un état $q_0$ de $M$ choisi, s'il existe un mot $u$ tel que pour tout état $q$ de $Q$, le chemin menant de $q$ à $q.u$ passe forcément par $q_0$.

*On transforme* $M$ *en une machine* $M'$ *en remplaçant toutes les transitions d'origine* $q_0$ *en boucles de* $q_0$ *vers* $q_0$ *:* $q_0$ *devient un état-puits.*
*q.u passe par* $q_0$ *dans* $M$ *si et seulement si* $\delta^*(q,u)$ *aboutit à* $q_0$ *dans* $M'$*. Ainsi* $\delta^*(q,u)$ *passe par* $q_0$ *dans* $M$ *pour tout* $q$ *implique que* $u$ *est synchronisant dans* $M'$*.*
*Inversement si* $u$ *est synchronisant dans* $M'$ *on a* $\delta^*(q,u) = q_1$ *dans* $M'$ *avec* $q_1$ *indépendant de* $q$*. Mais on a toujours* $\delta^*(q_0,u) = q_0$ *donc* $q_1 = q_0$ *: un mot synchronisant dans* $M'$ *aboutit à* $q_0$*.*
*On a ainsi ramené le problème à la recherche d'un mot synchronisant.*

# II Algorithmes classiques

On appellera *graphe d'automate* tout couple $(S, A)$ où $S$ est un ensemble dont les éléments sont appelés *sommets* et $A$ une partie de $S \times \Sigma \times S$ dont les éléments sont appelés *arcs*. Pour un arc $(q, x, q')$, $x$ est *l'étiquette* de l'arc, $q$ son *origine* et $q'$ son *extrémité*. Un graphe d'automate correspond donc à un automate non déterministe sans notion d'état initial ou final.

Par exemple, avec $\Sigma = a, b\}$, $S_0 = \{0, 1, 2, 3, 4, 5\}$ et

$$A_0 = \{(0,b,0),(0,a,3),(0,b,2),(0,a,1),(1,a,1),(1,a,2),(2,b,1),$$
$$(2,b,3),(2,b,4),(3,a,2),(4,a,1),(4,b,5),(5,a,1)\}$$

le graphe d'automate $G_0 = (S_0, A_0)$ est représenté en figure 4.

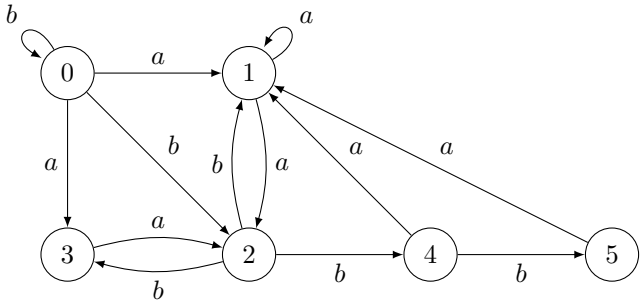



FIGURE 4 – Le graphe d'automate $G_0$

Soient $s$ et $s'$ deux sommets d'un graphe $(S, A)$.
Un chemin de $s$ vers $s'$ de longueur $\ell$ est une suite d'arcs $(s_1, x_1, s'_1), (s_2, x_2, s'_2), \ldots, (s_\ell, x_\ell, s'_\ell)$ de $A$ telle que $s_1 = s$, $s'_\ell = s'$ et pour tout $i$ de $[|1, \ell - 1|]$, $s'_i = s_{i+1}$.
L'étiquette de ce chemin est alors le mot $x_1 x_2 \ldots x_\ell$ et on dit que $s'$ est accessible depuis $s$. En particulier, pour tout $s \in S$, $s$ est accessible depuis $s$ par le chemin vide d'étiquette $\varepsilon$.
Dans les programmes à écrire, un graphe aura toujours pour ensemble de sommets un intervalle d'entiers $[|0, n-1|]$ et l'ensemble des arcs étiquetés par $\Sigma$ sera codé par un tableau de listes d'adjacences `v` : pour tout `s`$\in S$, `v.(s)` est la liste (dans n'importe quel ordre) de tous les couples $(s', x)$ tels que $(s, x, s')$ soit un arc du graphe. Pour des raisons de compatibilité ultérieure, les sommets (qui sont, rappelons-le, des entiers) seront codés par le type `etat`.
Ainsi, avec l'alphabet $\Sigma = \{a, b\}$, la lettre $a$ est codée 0 et la lettre $b$ est codée 1 ; l'ensemble des arcs du graphe $G_0$, dont chaque sommet est codé par son numéro, admet pour représentation

```
v0 : (etat*lettre) list array =[|[(0,1);(3,0);(2,1);(1,0)];
                                 [(1,0);(2,0)];
                                 [(1,1);(3,1);(4,1)];
                                 [(2,0)];
                                 [(1,0);(5,1)];
                                 [(1,0)] |]
```

On veut implémenter une file d'attente à l'aide d'un tableau circulaire. On définit pour cela le type `file` par

```
type 'a file ={tab : 'a array; mutable deb : int;
               mutable fin : int; mutable vide : bool}
```

- `deb` indique l'indice du premier élément dans la file,
- `fin` l'indice qui suit celui du dernier élément de la file,

- `vide` indiquant si la file est vide.

Les éléments sont rangés depuis la case `deb` jusqu'à la case précédent `fin` en repartant à la case 0 quand on arrive au bout du tableau (cf exemple). Ainsi, on peut très bien avoir l'indice `fin` plus petit que l'indice `deb`.
Dans la figure 5, la file contient les éléments $4, 0, 1, 12$ et 8 dans cet ordre, avec `fin`$= 2$ et `deb`$= 9$. On rappelle qu'un champ mutable peut voir sa valeur modifiée.

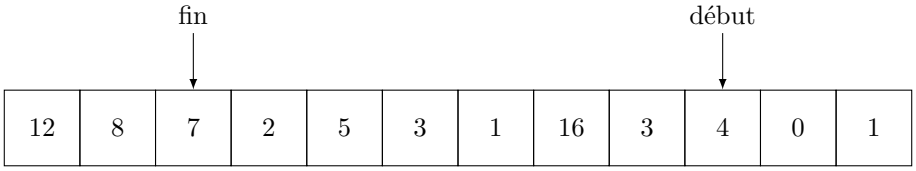



FIGURE 5 – Un exemple de file où `fin` $<$ `deb`

Par exemple, la syntaxe `f.deb` $\leftarrow 0$ affecte la valeur 0 au champ `deb` de la file `f`.

**Question 11** Écrire une fonction `ajoute : a file -> 'a -> unit` telle que `ajoute f x` ajoute `x` à la fin de la file d'attente `f`. Si c'est impossible, la fonction devra renvoyer un message d'erreur, en utilisant l'instruction `failwith "File pleine"`.

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

**Question 12** Écrire une fonction `retire : 'a file -> 'a` telle que `retire f` retire l'élément en tête de la file d'attente et la renvoie. Si c'est impossible, la fonction devra renvoyer un message d'erreur.

```
let retire f =
  if f.vide
  then failwith "La file est vide"
  else begin let n = Array.length f.tab in
       let y  = f.tab.(f.deb) in
       f.deb <- (f.deb + 1) mod n;
       if f.deb = f.fin then f.vide <- true;
       y end
```

**Question 13** Quelle est la complexité de ces fonctions ?

*Les deux fonctions ont une complexité constante donc en* $\mathcal{O}(1)$.

On considère l'algorithme A s'appliquant à un graphe d'automates $G = (S, A)$ et à un ensemble de sommets $E$ (on note $n = |S|$ et $\infty$, *vide* et *rien* des valeurs particulières).

**Algorithme 1 :** Algorithme A

```
F ← file d'attente vide
pour tous les s ∈ S faire
    D[s] ← ∞
    P[s] ← vide
c ← n
pour tous les s ∈ E faire
    insérer s à la fin de la file d'attente F
    D[s] ← 0
    P[s] ← rien
    c ← c − 1
tant que F n'est pas vide faire
    extraire le sommet s en tête de F
    pour tous les arcs (s, y, s') ∈ A tels que D[s'] = ∞ faire
        P[s'] ← (s, y)
        insérer s' à la fin de la file d'attente F
        c ← c − 1
retourner (c, D, P)
```

**Question 14** Justifier que l'algorithme A termine toujours.

*Dans le programme 1 la première boucle a une longueur déterminée, elle termine.*
*On ne met dans la file que les sommets tels que $D[s] = \infty$ et on modifie alors la valeurs de $D[s]$ : on ne devra insérer et extraire de la liste que $|S|$ sommets au plus.*
*Pour chacun de ces sommets extraites on parcourt leur liste de voisins donc on déterminera le résultat du test $D[s'] = \infty$ au plus $|A|$ fois.*
*Ainsi l'algorithme termine.*

**Question 15** Donner la complexité de cet algorithme en fonction de $|S|$ et $|A|$. On justifiera la réponse.

*Comme les opérations du programme se font en temps constant, les arguments ci-dessus montrent que la complexité est en $\mathcal{O}(|S| + |A|)$.*

**Question 16** Justifier qu'au début de chaque passage dans la boucle "**tant que** $F$ n'est pas vide", si $F$ contient dans l'ordre les sommets $s_1, s_2, \ldots, s_r$, alors $D[s_1] \leqslant D[s_2] \leqslant \ldots \leqslant D[s_r]$ et $D[S_r] - D[S_1] \leqslant 1$.

*On montre que la propriété suivante est un invariant de boucle.*
Si la file contient $s_1, s_2, \ldots, s_r$ alors il existe un entier $p$ et un indice $k$ vérifiant $1 \leqslant k \leqslant r$ tels que $D[s_1] = D[s_2] = \cdots = D[s_k] = p$ et $D[s_{k+1}] = \cdots = D[s_r] = p + 1$

- *Avant le premier passage dans la boucle* **tant que** *tous les éléments de la files sont les sommets de $E$ et vérifient $D[s] = 0$ : la propriété est vraie initialement avec $p = 0$ et $r = k = |E|$.*
- *Si elle est vrai avant un passage on enlève un élément de la file, sa valeur pour $D$ est $p$. On ajoute alors les sommets non visités dans la file avec $p + 1$ comme valeur de $D$.*
  *On revient alors avant la boucle* **tant que**
  - *si on avait $k > 1$ la propriété est vérifiée avec $k$ diminué de 1 et $p$ inchangé,*
  - *si on avait $k = 1$ alors soit la file est vide, soit elle ne contient que des sommets de valeur $p + 1$ pour $D$, la propriété est encore vérifiée avec $p$ remplacé par $p + 1$ et $k$ égal au nombre d'éléments dans la file..*

Pour $s$ sommet de $G$, on note $d_s$ la distance de $E$ à $s$ c'est à dire la longueur d'un plus court chemin d'un sommet de $E$ à $s$ (avec la convention $d_s = \infty$ s'il n'existe pas de tel chemin).

**Question 17** Justifier brièvement qu'à la fin de l'algorithme, pour tout sommet $s$, $D[s] \neq \infty$ si et seulement si $s$ est accessible depuis un sommet de $E$ et que $d_s \leq D[s]$.
Que désigne alors $c$ ?

---

*Chaque sommet dans la file est soit un élément de $E$ soit a été ajouté depuis un élément qui a été dans la file : on en déduit par récurrence sur le rang de sortie que tous les sommets visités sont accessibles depuis $E$.*
*Inversement si $s$ est un sommet accessible depuis $E$, il existe un chemin*
$s_0 \to s_1 \to \cdots \to s_k = s$ *avec* $s_0 \in E$.
$s_0$ *a été ajouté dans la file donc visité et, par récurrence sur $i$, tout sommet du chemin est voisin du sommet précédent dans le chemin donc a été ajouté, soit lors du traitement de $s_{i-1}$, soit auparavant s'il a une distance non infinie. En particulier $s$ a été visité.*
*Ainsi les sommets visités sont tous les sommets accessibles depuis les sommets de $E$ et uniquement ceux-là. Or $D[s] \neq \infty$ si et seulement si $s$ a été visité donc si et seulement si $s$ est accessible depuis un sommet de $E$.*
*$c$ est donc le nombre de sommets non accessibles depuis $E$.*
*De plus, la valeur de $D[s]$, pour $s$ accessible mais n'appartenant pas à $E$, vérifie $D[s] = D[s'] + 1$ quand il existe une arête de $s'$ à $s$. $D[s]$ est donc la longueur d'un chemin depuis un élément de $E$ vers $s$, elle est donc minorée par la distance minimale : $D[s] \geqslant d_s$.*

**Question 18** Montrer qu'en fait, à la fin, on a pour tout sommet $s$, $D[s] = d_s$.
Que vaut alors $P[s]$ ?

---

*On considère un chemin de longueur minimale de $s_0 \in E$ vers $s$ :*
$s_0 \to s_1 \to s_2 \to \cdots \to s_{p-1} \to s_p = s$. *En raison de la minimalité on a $d_{s_k} = k$.*
$s_{k+1}$ *est un voisin de $s_k$ donc il a est marqué (dans $D$) au plus tard lors la lecture de $s_k$ dans la file. En raison de la croissance de la lecture des valeurs de $D[s]$ on en déduit que* $D[s_{k+1}] \leqslant D[s_k] + 1$.
*On a donc* $D[s] = D[s_p] \leqslant D[s_0] + p = p = d_s$.
*Combinée avec $D[s] \geqslant d_s$ vue ci-dessus cette inégalité donne $D[s] = d_s$ pour tout sommet accessible ; si $s$ n'est pas accessible on a vu que $D[s] = \infty = d_s$.*
*Le chemin qui permet d'arriver à $s$ dans l'algorithme est donc un chemin minimal et $P[s]$ indique donc un prédécesseur de $s$ dans ce chemin (avec l'étiquette de la transition).*

**Question 19** Écrire une fonction `accessibles` prenant en entrée un graphe d'automate sous forme de son tableau de listes d'adjacence, `V : ((etat * lettre) list) array`, et un ensemble de sommets sous forme d'une liste d'états, `E : etat list` qui renvoie le triplet $(c, D, P)$ calculé selon l'algorithme précédent ses composantes ont pour type `int`, `int array` et `(etat * lettre) array`.
$\infty$, *vide* et *rien* seront respectivement codés par $-1$, $(-2, -1)$ et $(-1, -1)$ dans cette fonction.

---

*On transcrit le pseudo-code.*
***`infini`**, **`rien`** et **`vide`** ont été définies globalement.*

```
let infini = -1
let vide = (-2, -1)
let rien = (-1, -1)
```

*On traduit alors presque mot-à-mot le pseudo langage, la seule modification est le traitement du* **pour tout** $x \in E$ **tel que** *condition* **faire**
*que l'on transforme une une fonction de paramètre **`list`** appelée sur la liste à traiter. On commence par les créations*

```
let accessibles v ens  =
  let n = Array.length v in
  let tableau = Array.make matrix n n 0 in
  let f = {tab = tableau; deb = 0; fin = 0; vide = true} in
  let d = Array.make n infini in
  let p = Array.make n vide in
  let c = ref n in
```

*Viennent ensuite les initialisations pour chaque élément de* `ens`

```
let accessibles v ens  =
  ...
  let initialisation s =
    ajoute f s;
    d.(s) <- 0;
    p.(s) <- rien;
    incr c in
  List.iter initialisation ens;
```

*On peut alors effectuer le traitemen.*

```
let accessibles v ens  =
  ...
  List.iter initialisation ens;
  let rec traitement s (s1, y) =
    if d.(s1) = infini
    then begin
      d.(s1) <- d.(s) + 1;
      p.(s1) <- (s, y);
      ajoute f s1;
      c := !c - 1
    end in
  while not f.vide do
    let s = retire f in
    List.iter (traitement s) v.(s) done;
  !c, d, p
```

**Question 20** Écrire une fonction `chemin : etat -> (etat*lettre) array -> mot` qui, prenant en entrée un sommet `s` et le tableau `P` calculé à l'aide de la fonction `accessibles` sur un graphe $G$ et un ensemble $E$, renvoie un mot de longueur minimale qui est l'étiquette d'un chemin d'un sommet de $E$ à $s$ (ou un message d'erreur s'il n'en existe pas).

*Comme on détermine le sommet depuis la fin, on utilise une fonction récursive terminale.*

```
let chemin s p =
  if p.(s) =  vide
  then failwith "Le sommet n'est pas accessible"
  else let rec construire sommet mot =
         if p.(sommet) = rien  then mot
         else let (prec,y) = p.(sommet) in
              construire prec (y::mot) in
       construire s []
```

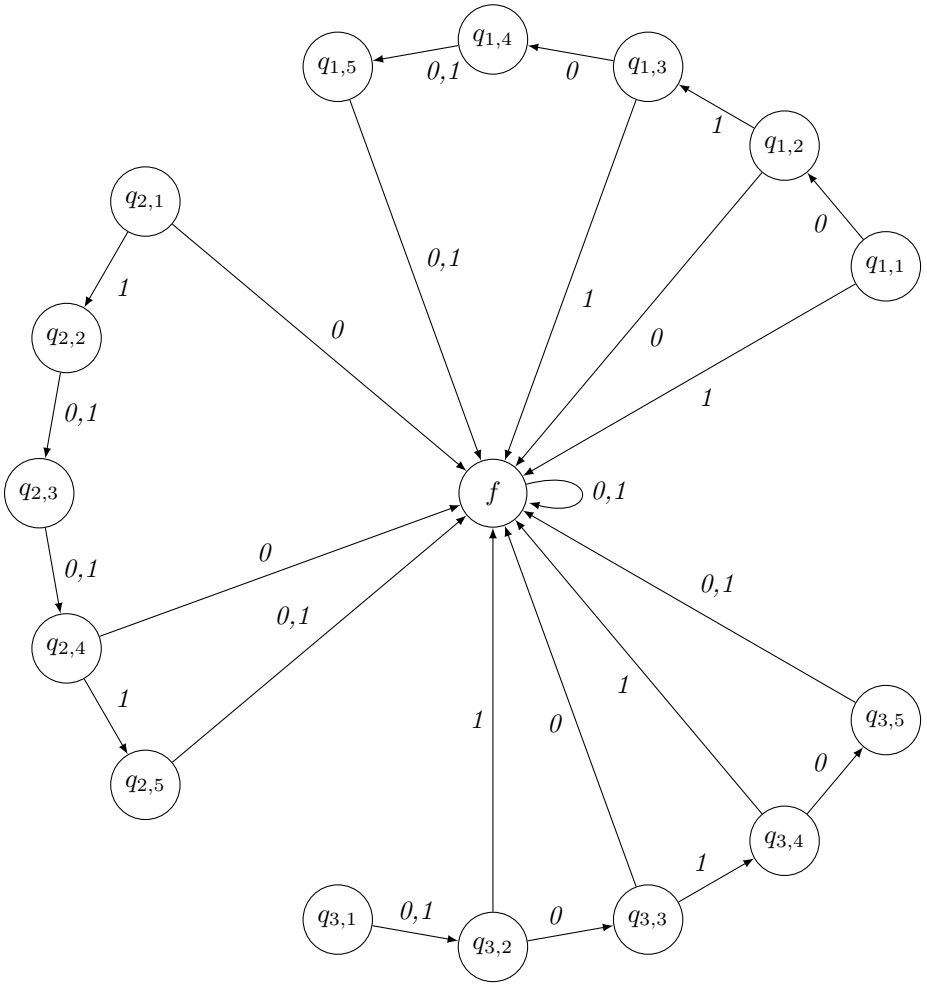
# III Reduction SAT

On s'intéresse dans cette partie à la satisfiabilité d'une formule logique portant sur des variables propositionnelles $x_1, \ldots, x_m$.
On note classiquement $\wedge$ le connecteur logique *et*, $\vee$ le connecteur *ou* et $\overline{f}$ la négation de $f$.
On appelle littéral une formule constituée d'une variable $x_i$ ou de sa négation $\overline{x_i}$, on appelle clause une disjonction de littéraux.
Considérons une formule logique sous forme normale conjonctive, c'est à dire sous la forme d'une conjonction de clauses.
Par exemple,$F_1 = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee x_4)$ est une formule sous forme normale conjonctive formée de trois clauses et portant sur quatre variables $x_1, x_2, x_3$ et $x_4$.
Soit $F$ une formule sous forme normale conjonctive, composée de $n$ clauses et faisant intervenir $m$ variables. On suppose les clauses numérotées $c_1, c_2, \ldots, c_n$. On veut ramener le problème de la satisfiabilité d'une telle formule au problème de la recherche d'un mot synchronisant de longueur $m$ au plus sur une certaine machine.
On introduit pour cela la machine suivante associée à $F$ :

- $Q$ est formé de $mn + n + 1$ états, un état particulier noté $f$ et $n(m + 1)$ autres états qu'on notera $q_{i,j}$ avec $(i, j) \in [|1, n|] \times [|1, m + 1|]$,
- $\Sigma = \{0, 1\}$,
- $f$ est un état puits, c'est à dire $\delta(f, 0) = \delta(f, 1) = f$,
- pour tout entier $i \in [|1, n|]$, $\delta(q_{i,m+1}, 0) = \delta(q_{i,m+1}, 1) = f$,
- pour tous $i \in [|1, n|]$ et $j \in [|1, m|]$,
  $\delta(q_{i,j}, 1) = f$ si le littéral $x_j$ apparaît dans la clause $c_i$ et $\delta(q_{i,j}, 1) = q_{i,j+1}$ sinon.
  $\delta(q_{i,j}, 0) = f$ si le littéral $\overline{x_j}$ apparaît dans la clause $c_i$ et $\delta(q_{i,j}, 1) = q_{i,j+1}$ sinon.

**Question 21** Représenter la machine associée à la formule $F_1$.

*16 sommets ...*




**Question 22** Donner une valuation $(v_1, v_2, v_3, v_4) \in [|0,1|]^4$ (la valeur $v_i$ étant associée à la variable $x_i$) satisfaisant $F_1$.
Le mot $v_1v_2v_3v_4$ est-il synchronisant ?

$(0,0,0,0)$ *est une valuation qui satisfait* $F_1$ *et* `0000` *est un mot synchronisant.*

**Question 23** Montrer que tout mot $u$ de longueur $m+1$ est synchronisant.
À quelle condition sur les $q_{i,1}.u$ un mot $u$ de longueur $m$ est-il synchronisant ?

*Chaque lettre envoie* $q_{i,j}$ *vers* $f$ *ou, pour* $j \leqslant m$, *vers* $q_{i,j+1}$ *et* $f$ *est stable par tout mot.*

- *Tout mot de longueur 1 envoie* $q_{i,m+1}$ *vers* $f$. *On en déduit que tout mot de longueur 1 ou plus envoie* $q_{i,m+1}$ *vers* $f$.
- *Tout mot de longueur 1 envoie* $q_{i,m}$ *vers* $f$ *ou* $q_{i,m+1}$. *Ainsi tout mot de longueur 2 ou plus envoie* $q_{i,m}$ *vers* $f$.
- *Tout mot de longueur 1 envoie* $q_{i,m-1}$ *vers* $f$ *ou* $q_{i,m}$. *D'où tout mot de longueur 3 ou plus envoie* $q_{i,m-1}$ *vers* $f$.
- *On poursuit jusqu'à* $q_{i,1}$ *qui est envoyé vers* $f$ *par tout mot de longueur* $m+1$ *ou plus.*

*Ainsi tout mot de longueur* $m+1$ *ou plus est synchronisant.*

*Si $u$ est de longueur $m$, on a $q_{i,k}.u = f$ pour $k \geqslant 2$. On a aussi $f.u = f$.*
*$u$ est donc synchronisant si et seulement si $q_{i,1}.u = f.$ pour tout $i$.*

**Question 24** Montrer que si la formule $F$ est satisfiable, toute valuation la satisfaisant donne un mot synchronisant de longueur $m$ pour l'automate.

---

*On suppose que $F$ est satisfiable par une valuation $(v_1, v_2, \ldots, v_m)$. On considère le mot $u = v_1 v_2 \cdots v_m$. Comme $u$ est de longueur $m$, d'après la question précédente, $u$ est donc synchronisant si et seulement si $q_{i,1}.u = f.$ pour tout $i$*
*Pour chaque clause $C_i$, il existe un littéral $l_k$ tel que sa valeur soit* **Vrai** *pour la valuation, ce qui signifie que $l_k = x_k$ et $v_k = 1$ ou $l_k = \overline{x_k}$ et $v_k = 0$.*
*On suppose que $k_0$ est le plus petit entier vérifiant cette propriété pour la clause $C_i$.*
*On a alors $q_{i,k_0}.v_{k_0} = f$ et, pour $k < k_0$,*

- *$\overline{x_k}$ n'apparaît pas dans la clause $C_i$ si $v_k = 0$*
- *ou $x_k$ n'apparaît pas dans la clause $C_i$ si $v_k = 1$,*
- *dans les deux cas $q_{i,k}.v_k = q_{i,k+1}$.*

*On peut donc calculer le chemin d'étiquette $u$ depuis $q_{i,1}$ :*
$q_{i,1}.u = q_{i,1}.v_1 v_2 \cdots v_m = q_{i,2}.v_2 \cdots v_m = \cdots = q_{i,k_0}.v_{k_0}.v_{k_0+1} \cdots v_m = f.v_{k_0+1} \cdots v_m = f.$
*Ainsi $q.u = f$ pour tout état $q$.*
*Tout mot associé à une valuation qui satisfait la formule est synchronisant.*

**Question 25** Inversement, prouver que si l'automate dispose d'un mot synchronisant de longueur $\leq m$, alors $F$ est satisfiable. Donner alors une valuation convenable.

---

*Inversement si $u$ est synchronisant pour l'automate avec $|u| \leqslant m$ alors, comme $f.u = f$, on doit avoir $q_{i,1}.u = f$ pour tout $i$. Cela implique qu'il existe un entier $k$ tel que $q_{i,k}.v_k = f$ donc que la valuation associée à $u$ satisfait toutes les clauses donc satisfait la formule. Si le mot est de longueur strictement inférieure à $m$ la valuation est prolongée de manière arbitraire pour les dernières variables.*

# IV Existence

On reprend dans cette partie le problème de l'existence d'un mot synchronisant pour une machine $M$.
Soit $M = (Q, \Sigma, \delta)$ une machine.
Pour toute partie $E$ de $Q$ et tout mot $u$ de $\Sigma^*$, on note $E.u = \{q.u,\ q \in E\}$.

**Question 26** Soit $u$ un mot synchronisant de $M$ et $u_0, u_1, \ldots, u_r$ une suite de préfixes de $u$ rangés dans l'ordre croissant de leur longueur et telle que $u_r = u$. Que peut-on dire de la suite des cardinaux $|Q.u_i|$ ?

---

*De manière générale, comme l'automate est déterministe $|E.x| \leqslant |E|$ pour tout ensemble de sommet $E$ et pour toute lettre $x$. On en déduit que $|Q.u_i|$ est décroissante avec, comme $u_r = u$ est synchronisant, $|Q.u_r| = 1$.*

**Question 27** Montrer qu'il existe un mot synchronisant si et seulement s'il existe pour tout couple d'états $(q, q')$ de $Q^2$ un mot $u_{q,q'}$ tel que $q.u_{q,q'} = q'.u_{q,q'}$.

---

*S'il existe un mot synchronisant $u$ alors $q.u = q'.u$ pour tous états $q$ et $q'$.*

*Inversement on suppose que, pour tout couple d'états $(q, q')$ il existe un mot $u_{q,q'}$ tel que $q.u_{q,q'} = q'.u_{q,q'}$.*
*On montre alors par récurrence sur $k = |E|$ qu'il existe un mot $u_E$ tel que $|E.u_E| = 1$ pour toute partie $E$ de $Q$. On dit que $E$ est* **synchronisable**.
*Pour $|E| = 1$ tout mot convient.*
*On suppose que tout ensemble de cardinal au plus $k$ est synchronisable avec $1 \leqslant k < m$.*
*Si $E$ est de cardinal $k+1$ il a deux élément distincts $q_1$ et $q_2$. On considère $u_1$ tel que $q_1.u_1 = q_2.u_1$. L'ensemble $E' = E.u_1$ est donc de cardinal $k$ au plus donc il existe un mot $u_{E'}$ tel que $E'.u_{E'}$ est un singleton par hypothèse de récurrence.*
*On a alors $E.(u_1.u_{E'}) = (E.u_1).u_{E'} = E'.u_{E'}$ singleton donc on peut choisir $u_E = u_1.u_{E'}$.*
*On a alors, pour tout $q \in E$, $q.u_1u_2 = q'.u_2 = q_0$ car $q' = q.u_1 \in E'$ : $E$ est synchronisable.*
*Tout ensemble de cardinal au plus $k+1$ est synchronisable.*
*La récurrence est prouvée.*
*En particulier $Q$ est synchronisable donc $u_Q$ est un mot synchronisant.*

On veut se servir du critère établi ci-dessus pour déterminer s'il existe un mot synchronisant. Pour cela, on associe à la machine $M$ la machine $\widetilde{M} = (\widetilde{Q}, \Sigma, \widetilde{\delta})$ définie par :

- $\widetilde{Q}$ est formé des parties à un ou deux éléments de $Q$ ;
- $\widetilde{\delta}$ est définie par $\forall (E, x) \in \widetilde{Q} \times \Sigma,\ \widetilde{\delta}(E) = \{\delta(q, x),\ q \in E\}$.

**Question 28** Si $n = |Q|$, que vaut $\widetilde{n} = |\widetilde{Q}|$ ?

---

$$|\widetilde{Q}| = \binom{n}{1} + \binom{n}{2} = \frac{n(n+1)}{2}.$$

On a dit que pour la modélisation informatique, l'ensemble d'états d'une machine doit être modélisée par un intervalle $[|0, n-1|]$. $\widetilde{Q}$ doit donc être modélisé par l'intervalle $[|0, \widetilde{n}-1|]$.
Soit $\varphi_n$ une bijection de $\widetilde{Q}$ sur $[|0, \widetilde{n}-1|]$.
On suppose qu'on dispose d'une fonction `set_to_nb : int -> (etat list) -> etat` telle que `set_to_nb n l` pour `n` représentant un élément $n \in \mathbb{N}^*$ et `l` représentant une liste $\ell$ d'états renvoie $\varphi_n(\{i\})$ si $\ell = [i]$ avec $0 \leq i \leq n-1$ et $\varphi_n(\{i, j\})$ si $\ell = [i; j]$ avec $0 \leq i < j \leq n-1$.
On suppose définie aussi la fonction réciproque `nb_to_set : int -> etat -> (etat list)` telle que `nb_to_set n q` pour $n \in \mathbb{N}^*$ et $q \in [|0, \widetilde{n}-1|]$ renvoie une liste d'états de la forme $[i]$ ou $[i; j]$ (avec $i < j$) correspondant à $\varphi_n^{-1}(q)$.
Ces deux fonctions de conversion sont supposées agir en temps constant.

Enfin, pour ne pas confondre un état de $\widetilde{Q}$ avec sa représentation informatique par un entier, on notera $\overline{q}$ l'entier associé à l'état $q$.

**Question 29** Écrire une fonction `delta2 : machine -> etat -> lettre -> etat` qui prenant en entrée une machine $M$, un état $\overline{q}$ de $\widetilde{Q}$ et une lettre $x$, renvoie l'état de $\widetilde{Q}$ atteint en lisant la lettre $x$ depuis l'état $q$ dans $\widetilde{M}$.

```
let delta2 mch q x =
  let n = mch.n_etats in
  let ens = nb_to_set n q in
  match ens with
  |[i] -> set_to_nb n [mch.delta i x]
  |[i; j] -> let i1 = mch.delta i x in
             let j1 = mch.delta j x in
             if i1 = j1
             then set_to_nb n [i1]
             else if i1 < j1
                  then set_to_nb n [i1; j1]
                  else set_to_nb n [j1; i1]
  |_ -> failwith "Ceci ne devrait pas arriver"
```

Il est clair qu'à la machine $\widetilde{M}$, on peut associer un graphe d'automate $\widetilde{G}$ dont l'ensemble des sommets est $\widetilde{Q}$ et dont l'ensemble des arcs est $\{(q, x, \widetilde{\delta}(q, x)),\ (q, x) \in \widetilde{Q} \times \Sigma\}$.
On associe alors à $\widetilde{G}$ le graphe retourné $\widetilde{G}_R$ qui a les mêmes sommets que $\widetilde{G}$ mais dont les arcs sont retournés (i.e. $(q, x, q')$ est un arc de $\widetilde{G}_R$ si et seulement si $(q', x, q)$ est un arc de $\widetilde{G}$).

**Question 30** Écrire une fonction `retourne_machine : machine -> ((etat*lettre) list) array` qui à partir d'une machine $M$, calculer le tableau $V$ des listes d'adjacence du graphe $\widetilde{G}_R$.

```
let retourne_machine mch =
  let n = mch.n_etats in
  let ntilde = n*(n+1)/2 in
  let v = Array.make ntilde [] in
  for i = 0 to (n-1) do
    for x = 0 to (mch.n_lettres -1) do
      let q = set_to_nb n [i] in
      let q1 = delta2 mch q x in
      v.(q1) <- (q, x)::(v.(q1));
      for j = (i+1) to (n-1) do
        let q = set_to_nb n [i; j] in
        let q1 = delta2 mch q x in
        v.(q1) <- (q, x)::(v.(q1))
      done
    done
  done;
  v
```

**Question 31** Justifier qu'il suffit d'appliquer la fonction `accessibles` de la partie **2** au graphe $\widetilde{G}_R$ et à l'ensemble des sommets de $\widetilde{G}_R$ correspondant à des singletons pour déterminer si la machine $M$ possède un mot synchronisant.

*D'après la question 27, 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 envoie cet état vers $[k]$.*

*Comme la transformée d'un singleton est toujours un singleton, la condition ci-dessus est que tout état de $\widetilde{M}$ peut être envoyé dans un singleton. Ceci revient à voir si, dans $\widetilde{G}_R$, on peut atteindre tout état depuis l'ensemble des singletons.*
*Il suffit donc d'appliquer* `accessibles` *depuis l'ensemble des états singleton dans $\widetilde{G}_R$ et de tester si le nombre d'éléments non atteints est 0.*

**Question 32** Écrire une fonction `existe_synchronisant : machine -> bool` qui dit si une machine possède un mot synchronisant.

---

*On commence par la liste des singletons (le résultat est "à l'envers").*

```
let rec singletons n =
  if n =  0
  then []
  else [n-1] :: (singletons (n-1)
```

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

*Jan Černỳ, chercheur slovaque, a conjecturé au milieu des années 60 que si une machine à $n$ états possédait un mot synchronisant, elle en avait un de longueur $\leq (n-1)^2$.*
*La construction faite dans la partie 3 affirme que la recherche, dans une machine, d'un mot synchronisant de longueur $\leq m$ fixé est au moins aussi difficile en terme de complexité que celui de la satisfiabilité d'une formule logique à $m$ variables sous forme normale conjonctive.*