Concours blanc

17 décembre 2024

INFORMATIQUE MPI

DURÉE DE L'ÉPREUVE : **4 heures**

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Ce sujet comporte onze pages numérotées de 1/21 à 21/21

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 4 parties indépendantes, utilisant les langages de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

- La partie I s'intéresse à des généralités sur les machines, et à leur implémentation en C.
- La partie II s'intéresse à une implémentation en C des files par des listes chaînées circulaires.
- La partie III s'intéresse à des algorithmes classiques sur les graphes, transposé aux graphes d'automates.
- La partie IV s'intéresse à la difficulté d'un problème de décision associé à la recherche de mot synchronisant.
- La partie V propose un algorithme permettant de déterminer si une machine possède un mot synchronisant.
- Enfin, la partie VI étudie des jeux d'accessibilité à 2 joueurs sur des machines non-déterministes.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus. En langage OCaml, on autorisera toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`).

Sans précision supplémentaire, lorsqu'une question demande la complexité d'une fonction, il s'agira de la complexité temporelle dans le pire des cas. La complexité sera exprimée sous la forme $\mathcal{O}(f(n,m))$ où n et m sont les tailles des arguments de la fonction, et f une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

On appelle **machine** un triplet (Q, Σ, δ) où Q est un ensemble fini non vide d'**états**, Σ un alphabet et δ une application de $Q \times \Sigma$ dans Q appelée **fonction de transition**. Une machine peut donc être vue comme un automate fini déterministe complet sans notion d'état initial ou final. Comme pour les automates finis, on utilisera la notion de fonction de transition étendue définie par :

— pour tout $q \in Q$, $\delta^*(q, \varepsilon) = q$;
— pour tous $q \in Q$, $u \in \Sigma^*$ et $a \in \Sigma$, $\delta^*(q, ua) = \delta(\delta^*(q, u), a)$.

Pour deux états q et q', q' est dit **accessible** depuis q s'il existe un mot $u \in \Sigma^*$ tel que $\delta^*(q, u) = q'$.
Un mot $u \in \Sigma^*$ est dit **synchronisant** pour une machine (Q, σ, δ) s'il existe $q_0 \in Q$ tel que $\forall q \in Q$, $\delta^*(q, u) = q_0$.
L'existence de tels mots permet de ramener une machine dans un état particulier connu en lisant un mot donné, donc en pratique de réinitialiser une machine réelle. La figure 1 représente une machine M_0. On pourra remarquer que ba et bb sont des mots synchronisants pour M_0.

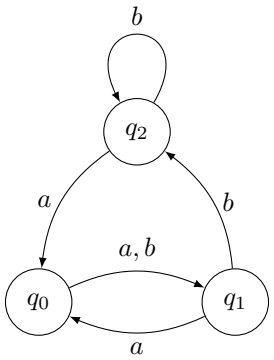


FIGURE 1 – La machine M_0.

Partie I. Considérations générales – 30 pts

On représente une lettre de Σ en C par un objet de type `char`. On supposera que Σ est constitué de lettres minuscules consécutives de l'alphabet courant commençant par la lettre `'a'`. Ainsi, si `a` est un objet de type `char`, alors `a - 97` est un entier compris entre 0 et 25. Un mot de Σ^* sera alors représenté par une chaîne de caractères de type `char*`. On rappelle que le caractère de fin de chaîne est `'\0'`.
Une machine sera représentée par une valeur de type `machine` défini par :

```
struct Machine{
    int tQ;
    int tSigma;
    int** delta;
};

typedef struct Machine machine;
```

tel que si $M = (Q, \Sigma, \delta)$ est une machine représenté par un pointeur `M` vers une valeur de type `machine`, alors :

— `M->tQ` représente $|Q|$, on suppose $Q = [\![0, |Q| - 1]\!]$;
— `M->tSigma` représente $|\Sigma|$, on suppose $\Sigma = [\![0, |\Sigma| - 1]\!]$;
— `M->delta` correspond à un tableau des transitions, c'est-à-dire tel que si $q \in Q$ et $a \in \Sigma$, alors `M->delta[q][a]` vaut $\delta(q, a)$.

☐ 1 – Écrire une fonction `machine* init_machine(int tQ, int tSigma)` qui crée une machine telle que $|Q|$ et $|\Sigma|$ sont donnés en arguments et, pour tout $q \in Q$ et $a \in \Sigma$, $\delta(q, a) = q$.

Réponse 1.

C
```c
machine* init_machine(int tQ, int tSigma){
  machine *M = malloc(sizeof(machine));
  M->tQ = tQ; M->tSigma = tSigma;
  M->delta = malloc(tQ * sizeof(int *));
  for (int q = 0; q < tQ; q = q + 1){
    M->delta[q] = malloc(tSigma * sizeof(int));
    for (int a = 0; a < tSigma; a = a + 1){
      M->delta[q][a] = q;
    }
  }
}
```

☐ 2 – Écrire une fonction `void liberer_machine(machine* M)` qui libère l'espace mémoire occupé par une machine.

Réponse 2.

C
```c
void liberer_machine(machine* M){
  if (M == NULL) { return; }
  if (M->delta != NULL) {
    for (int q = 0; q < M->tQ; q = q + 1){
      if (M->delta[q] != NULL) { free(M->delta[q]); }
      }
      free(M->delta);
  }
  free(M);
}
```

☐ 3 – Que dire de l'ensemble des mots synchronisant pour une machine ayant un seul état ?

Réponse 3. Si $\Sigma \neq \emptyset$, alors tout mot non vide est synchronisant pour une machine à un seul état.

Dans toute la suite du problème, on supposera que les machines ont au moins deux états.

☐ 4 – On considère la machine M_1 représentée figure 2, sur l'alphabet $\Sigma = \{a\}$. Donner un mot synchronisant pour M_1 s'il en existe un. Justifier la réponse.

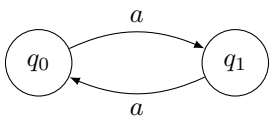


FIGURE 2 – La machine M_1.

Réponse 4. On remarque que si $|u| \equiv 0[2]$, alors $\delta^*(q_i, u) = q_i$ et si $|u| \equiv 1[2]$, alors $\delta^*(q_i, u) = q_{1-i}$. On en déduit que pour tout mot u, $\delta^*(q_0, u) \neq \delta^*(q_1, u)$, donc qu'il n'existe pas de mot synchronisant.

☐ 5 – Donner un mot synchronisant de trois lettres pour la machine M_2 représentée figure 3. On ne demande pas de justifier la réponse.

Réponse 5. On remarque que la lecture d'un a emmène à l'état q_1 ou l'état q_2. De plus, la lecture de da depuis l'un de ces deux états emmène nécessairement vers l'état q_1. On en déduit que ada est un mot synchronisant pour M_2.

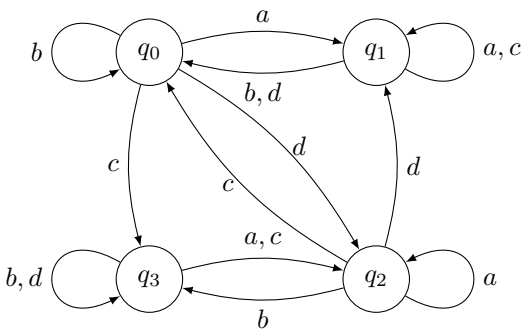


FIGURE 3 – La machine M_2.

☐ 6 – Écrire une fonction `int delta_etoile(machine* M, int q, char* u)` qui prend en arguments une machine $M = (Q, \Sigma, \delta)$, un état q et un mot u et renvoie $\delta^*(q, u)$. On prendra garde que `u` est une chaîne de caractères, et qu'un caractère a une valeur comprise entre 97 et 122 et non entre 0 et 25.

Réponse 6.

C
```c
int delta_etoile(machine* M, int q, char* u){
  for (int i = 0; u[i] != '\0'; i = i + 1)
    { q = M->delta[q][u[i] - 97]; }
  return q;
}
```

☐ 7 – Écrire une fonction `bool synchronisant(machine* M, char* u)` qui prend en argument une machine M et un mot u et renvoie le booléen `true` si u est synchronisant pour M et `false` sinon.

Réponse 7.

C
```c
bool synchronisant(machine* M, char* u){
  int qq = delta_etoile(M, 0, u);
  for (int q = 1; q < M->tQ; q = q + 1){
    if (delta_etoile(M, q, u) != qq) { return false; }
  }
  return true;
}
```

☐ 8 – Montrer que si une machine admet un mot synchronisant alors il existe $a \in \Sigma$ et $q \neq q' \in Q$ tels que $\delta(q, a) = \delta(q', a)$.

Réponse 8. Soit M une machine à au moins deux états admettant un mot synchronisant $u = u_1 \dots u_k$ où les $u_i \in \Sigma$. Soient $q \neq q'$ deux états de M. Alors $\delta^*(q, u) = \delta^*(q', u)$. Notons $q = q_0 \xrightarrow{u_1} q_1 \xrightarrow{u_2} \dots \xrightarrow{u_k} q_k = \delta^*(q, u)$ et $q' = q'_0 \xrightarrow{u_1} q'_1 \xrightarrow{u_2} \dots \xrightarrow{u_k} q'_k = q_k$ des calculs de u depuis les états q et q'. Comme $q_0 \neq q'_0$ et $q_k = q'_k$, on en déduit qu'il existe $i \in [\![0, k-1]\!]$ minimal tel que $q_i \neq q'_i$ et $q_{i+1} = q'_{i+1}$. Dès lors, on a $\delta(q_i, u_{i+1}) = \delta(q'_i, u_{i+1})$.

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 $\widehat{Q} = \mathcal{P}(Q)$ et $\widehat{\delta}$ est définie par :

$$\forall P \subset Q, \forall a \in \Sigma, \widehat{\delta}(P, a) = \{\delta(p, a), p \in P\}$$

☐ 9 – Montrer 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) de $\widehat{M}$.

Réponse 9. On remarque que s'il existe un mot synchronisant u pour M, tel que $\forall q \in Q$, $\delta^*(q,u) = q_0$, alors $\widehat{\delta}^*(Q,u) = \{q_0\}$. On vérifie aisément que la réciproque est vraie. 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 $\widehat{M}$.

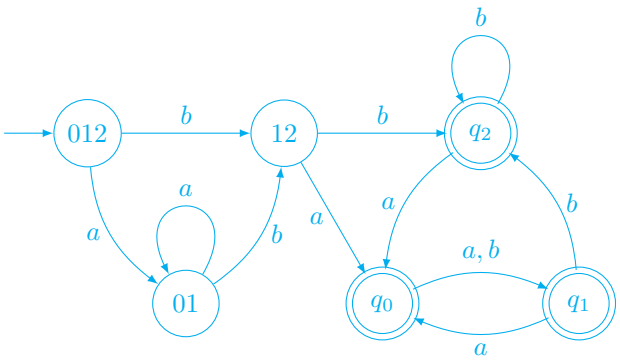
☐ 10 – En déduire que le langage $LS(M)$ des mots synchronisants de M est reconnaissable.

Réponse 10. On définit l'automate déterministe $(\Sigma, \widehat{Q}, \widehat{\delta}, Q, \{\{q\}, q \in Q\})$. D'après la proposition précédente, cet automate reconnaît bien l'ensemble de tous les mots synchronisants de M. On en déduit que $LS(M)$ est reconnaissable.

☐ 11 – Déterminer puis représenter graphiquement un automate fini déterministe (pas nécessairement complet) reconnaissant $LS(M_0)$.

Réponse 11. On détermine l'automate des parties sous forme de tableau, puis on le représente en enlevant les états non utiles (l'état initial étant Q) :

	a	b
$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_1, q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	$\{q_0\}$	$\{q_2\}$
$\{q_0\}$	$\{q_1\}$	$\{q_1\}$
$\{q_1\}$	$\{q_0\}$	$\{q_2\}$
$\{q_2\}$	$\{q_0\}$	$\{q_2\}$



Notons qu'on peut simplifier cet automate en fusionnant q_0, q_1 et q_2 d'une part et 012 et 01 d'autre part.

☐ 12 – 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 $\delta^*(q,u)$ passe forcément par q_0.

Réponse 12. Soit $M = (Q, \Sigma, \delta)$ une machine et q_0 un état de M. On pose $M' = (Q, \Sigma, \delta')$ telle que :
- $\forall q \in Q \setminus \{q_0\}$, $\forall a \in \Sigma$, $\delta'(q,a) = \delta(q,a)$;
- $\forall a \in \Sigma$, $\delta'(q_0, a) = q_0$.

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

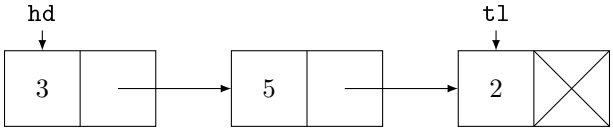
Partie II. Files – 20 pts

Dans cette partie, on s'intéresse à l'implémentation de la structure de donnée abstraite file par une liste chaînée circulaire.
Voici un exemple d'implémentation en C de la structure de file par une liste de maillons chaînés, `hd` pointant vers le maillon de tête (prochain à sortir de la file) et `tl` pointant vers le maillon de queue (dernier entré) :

```c
struct maillon {
  int data;
  struct maillon *next;
};

struct file {
  struct maillon *hd;
  struct maillon *tl;
};
```

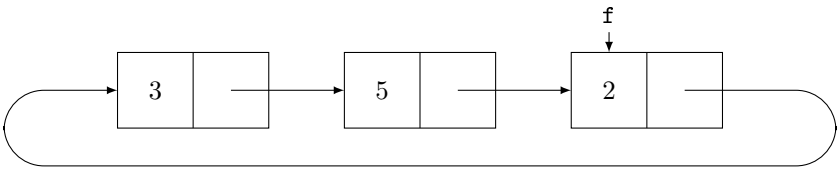
Le champ `next` d'un maillon pointe vers le maillon suivant, vers la queue de la file. Voici un exemple de représentation d'une file dans laquelle on a enfilé les valeurs 3, 5, et 2 dans cet ordre.



Le champ `next` du maillon de queue vaut donc toujours `NULL`. Fort de cette observation, nous décidons de remplacer la valeur du champ `next` du maillon pointé par `tl` par le pointeur `hd`. La file peut alors n'être représenté que par le pointeur `tl` si elle est non vide, et par le pointeur `NULL` sinon :

```c
typedef struct maillon *file;
```

Si la file de l'exemple précédent est contenu dans la variable `f` de type `file`, on a alors le schéma suivant :



☐ 13 – Écrire une fonction `bool est_vide(file f)` qui teste si une file est vide. La complexité temporelle de cette fonction doit être $O(1)$ dans le pire des cas.

Réponse 13.

C

```c
bool est_vide(file f){
  return f == NULL;
}
```

☐ 14 – Dans les questions qui suivent, on fera bien attention à ne pas déréférencer de pointeur `NULL`.
Écrire une fonction `file enfile(int x, file f)` qui ajoute l'élément x à la file f. La fonction renvoie le nouveau pointeur désignant la file. La complexité temporelle de cette fonction doit être $O(1)$ dans le pire des cas.

☐ 15 – Écrire une fonction `file enfile(int x, file f)` qui ajoute l'élément x à la file f. La fonction renvoie le nouveau pointeur désignant la file. La complexité temporelle de cette fonction doit être $O(1)$ dans le pire des cas.

Réponse 15.

C
```c
file enfile(int x, file f){
  struct maillon *m = malloc(sizeof(struct maillon));
  m->data = x;
  if (est_vide(f)){
    m->next = m;
    return m;
  }
  m->next = f->next;
  f->next = m;
  return m;
}
```

☐ 16 – Écrire une fonction `int peek(file f)` qui renvoie l'élément en tête de la file f sans la modifier. La complexité temporelle de cette fonction doit être $O(1)$ dans le pire des cas.

Réponse 16.

C
```c
int peek(file f){
  assert(!est_vide(f));
  return f->next->data;
}
```

☐ 17 – Écrire une fonction `file defile(file f)` qui enlève l'élément de tête de la file et libère la mémoire si nécessaire. La fonction renvoie le nouveau pointeur désignant la file. La complexité temporelle de cette fonction doit être $O(1)$ dans le pire des cas.

Réponse 17.

C
```c
file defile(file f){
  assert(!est_vide(f));
  if (f->next == f) { free(f); return NULL; }
  file tmp = f->next;
  f->next = tmp->next;
  free(tmp);
  return f;
}
```

☐ 18 – Écrire une fonction `void affiche(file f)` qui affiche tous les éléments de la file f. La file f ne doit pas être modifiée. La complexité temporelle de cette fonction doit être $O(n)$ dans le pire des cas, où n est le nombre d'éléments dans la file.

Réponse 18.

```C
void affiche(file f){
  if (f == NULL) { return; }
  printf("%d", f->data);
  file tmp = f-> next;
  while (tmp != f){
    printf(",%d",tmp->data);
    tmp = tmp->next;
  }
  printf("\n");
}
```

☐ 19 – Si on respecte l'invariant posé en début de section, la file est représentée par une chaîne de maillons qui forme un cycle. Toute chaîne de maillon ne forme pas nécessairement un cycle : elle peut être terminée (comme dans les cas des listes finies), ou encore contenir un cycle, mais qui ne contient pas le premier maillon.
Proposer un algorithme permettant de détecter si un pointeur correspond bien à une file, c'est-à-dire une liste chaînée circulaire, ou non. Évaluer sa complexité en temps et en espace dans le pire des cas en fonction de n, le nombre de maillons accessibles depuis le pointeur donné en entrée.
On ne demande pas d'écrire cet algorithme en langage C.

☐ 20 – Si on respecte l'invariant posé en début de section, la file est représentée par une chaîne de maillons qui forme un cycle. Toute chaîne de maillon ne forme pas nécessairement un cycle.

Donner les 3 cas qui peuvent survenir si on suit une chaîne de maillons, et écrire une fonction `bool is_correct(file f)` qui renvoie `true` si `f` pointe vers une file, et `false` sinon. Cette fonction doit avoir une complexité temporelle dans le pire des cas en $O(n)$, où n est le nombre de maillons accessibles depuis `f` (il existe une solution qui n'utilise comme mémoire temporaire que $O(1)$).

Réponse 20. Avec 2 pointeurs qui avancent à vitesses différentes. Si le pointeur le plus rapide revient au point de départ, le pointeur donné correspond bien à une file. Si il rencontre un pointeur NULL, on a une liste chainée. Si les 2 pointeurs se rencontrent, il y a une boucle qui ne contient pas le maillon initial.

```C
bool is_correct(file f){
  file tortue = f;
  file lievre = f;
  while (lievre != f && lievre != NULL){
    lievre = lievre-> next;
    if (lievre == tortue) { return false; }
    lievre = lievre-> next;
    if (lievre == tortue) { return false; }
    tortue = tortue -> next;
  }
  return lievre = f;
}
```

Partie III. Algorithmes classiques – 20 pts

On appellera **graphe d'automate** tout couple (S, A) où S est un ensemble dont les éléments sont appelés **sommets** et A est une partie de $S \times \Sigma \times S$ dont les éléments sont appelés **arcs**. Pour un arc (q, a, q'), a 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. Pour simplifier la rédaction et éviter l'écriture lourde de

`int_of_char c - int_of_char 'a'` à multiple reprises, nous supposerons désormais que les lettres sont données en tant qu'entiers entre 0 et $|\Sigma| - 1$, le a correspondant au 0.

Par exemple, avec $\Sigma = \{a, b\}$, $S_0 = \{0, 1, 2, 3, 4, 5\}$ et A_0 défini par :

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

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

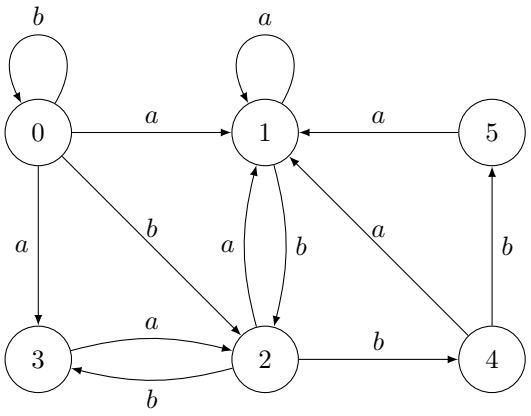


FIGURE 4 – Le graphe d'automate G_0.

Un graphe aura toujours pour ensemble de sommets un intervalle d'entiers $[\![0, n-1]\!]$ et l'ensemble des arcs étiquetés par Σ (comme précédemment supposé être un intervalle $[\![0, |\Sigma| - 1]\!]$) sera codé par un tableau de listes d'adjacences `g` : pour tout $s \in S$, `g.(s)` est la liste (dans un ordre arbitraire) de tous les couples (t, a) tel que $(s, a, t) \in E$. Par exemple, le graphe G_0 est ainsi représenté par :

```
type graphe = (int * int) list array;;

let g0 = [| [(1, 0); (3, 0); (0, 1); (2, 1)];
            [(1, 0); (2, 0)];
            [(1, 1); (3, 1); (4, 1)];
            [(2, 0)];
            [(1, 0); (5, 1)];
            [(1, 0)] |];;
```

On considère l'algorithme ci-dessous s'appliquant à un graphe d'automate $G = (S, A)$ et à un ensemble de sommets X (on note $n = |S|$, ∞, *vide* et *rien* des valeurs particulières).

Entrée : Graphe d'automate $G = (S, A)$ et $X \subset S$
Début algorithme
  $F \leftarrow$ file vide
  $D \leftarrow$ tableau de taille n contenant les valeurs ∞
  $P \leftarrow$ tableau de taille n contenant les valeurs *vide*
  $c \leftarrow n$
  Pour $s \in X$ **Faire**
    Insérer s à F
    $D[s] \leftarrow 0$
    $P[s] \leftarrow$ *rien*
    $c \leftarrow c - 1$
  Tant que $F \neq \emptyset$ **Faire**
    $s \leftarrow$ extraction de F
    Pour $(s, a, t) \in A$ tel que $D[t] = \infty$ **Faire**
      $D[t] \leftarrow D[s] + 1$
      $P[t] \leftarrow (s, a)$
      Insérer t à F
      $c \leftarrow c - 1$
  Renvoyer (c, D, P)

☐ 21 – Justifier que l'algorithme termine toujours.

Réponse 21. La seule problématique à étudier pour montrer la terminaison est la terminaison de la boucle **Tant que**. Pour s'assurer que cette boucle termine, il suffit de remarquer qu'un sommet v n'est ajouté à la file qu'au plus une fois : soit pendant la première boucle **Pour**, soit pendant la boucle **Pour** à l'intérieur de la boucle **Tant que**. Dans cette dernière, un sommet n'est ajouté que si $D[v] = \infty$, mais la valeur de $D[v]$ est ensuite modifiée et ne prendra plus la valeur ∞.

Plus formellement, un variant de boucle est le couple $(|\{v|D[v] = \infty\}|, |F|)$ qui diminue strictement pour l'ordre lexicographique, et $|\{v|D[v] = \infty\}|$ est exactement la valeur de c.

☐ 22 – Déterminer la complexité de cet algorithme en fonction de $n = |S|$ et $p = |A|$.

Réponse 22. On reconnaît ici un algorithme de parcours en largeur. La première boucle **Pour** est de taille au plus n, et les opérations qui y sont faites sont en temps constant. Dans la boucle **Tant que**, on ne fera les opérations de modifications de valeurs qu'au plus une fois pour chaque arête. On en déduit une complexité en $\mathcal{O}(n + p)$.

☐ 23 – Justifier qu'au début de chaque passage dans la boucle « **Tant que** $F \neq \emptyset$ », si F contient dans l'ordre où ils ont été ajoutés les sommets $s_1, \ldots, s_r$, alors $D[s_1] \leq D[s_2] \leq \ldots \leq D[s_r]$ et $D[s_r] - D[s_1] \leq 1$.

Réponse 23. C'est un invariant de la boucle « **Tant que** $F \neq \emptyset$ » :
- Initialement, la file contient les éléments de W, et $D[v] = 0$ pour $v \in W$.
- Supposons la propriété vraie au k-ème passage dans la boucle, et la file composée des éléments $v_1, \ldots v_r$. Après le passage dans la boucle, la file sera composée des éléments $v_2, \ldots, v_r, v'_1, \ldots, v'_s$ et on aura $D[v'_i] = D[v_1] + 1$. De plus, comme $D[v_r] \leq D[v_1] + 1$ (par hypothèse), alors on aura bien $D[v_2] \leq \ldots \leq D[v_r] \leq D[v'_1] \leq \ldots \leq D[v'_s]$. De plus, si $r \geq 2$, alors $D[v'_s] - D[v_2] = D[v_1] + 1 - D[v_2] \leq 1$ (car $D[v_1] \leq D[v_2]$). Sinon, $D[v'_s] - D[v'_1] = 0$. L'invariant est bien conservé.

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

☐ 24 – Justifier 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 X et que $d_s \leq D[s]$. Que désigne alors c ?

Réponse 24. ($\Rightarrow$) Supposons $D[v] \neq \infty$ et montrons que v est accessible depuis W et que $D[v] \geq d_v$, par récurrence sur $D[v]$.
Si $D[v] = 0$, alors $v \in W$, donc le résultat est prouvé (car $d_v = 0$). Sinon, supposons le résultat vrai pour tout v' tel que $D[v'] \leq k \in \mathbb{N}$ et soit $v \in V$ tel que $D[v] = k + 1$. D'après l'algorithme, v a un voisin v' tel que $D[v'] = k$, et par hypothèse de récurrence, v' est accessible depuis W. On en déduit que v est accessible depuis W. De plus, comme $D[v'] \geq d_{v'}$, on a $d_v \leq d_{v'} + 1 \leq D[v'] + 1 = D[v]$.

($\Leftarrow$) Supposons que v est accessible depuis W et montrons que $D[v] \neq \infty$, par récurrence sur d_v.
Si $d_v = 0$, alors $v \in W$ et $D[v] = 0 \neq \infty$. Sinon, supposons le résultat vrai pour tout v' tel que $d_{v'} \leq k \in \mathbb{N}$. Soit $v \in V$ tel que $d_v = k + 1$. Alors il existe un chemin de taille $k + 1$ de la forme $v_0 \rightsquigarrow v' \to v$ où $\rightsquigarrow$ désigne un chemin de taille k et $\to$ désigne une arête, avec $v_0 \in W$. Par hypothèse de récurrence, on a $D[v'] \neq \infty$, car $d_{v'} = k$. Dès lors, sachant que v' passe par la file, quand v' est défilé, soit $D[v] \neq \infty$ et le résultat est prouvé, soit $D[v] = \infty$ et on pose $D[v] = D[v'] + 1 \neq \infty$.

La valeur de c correspond au nombre de sommets non accessibles par le parcours (elle commence à n et diminue de 1 chaque fois qu'un sommet non accessible est atteint).

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

Réponse 25. Supposons par l'absurde qu'il existe $v \in V$ tel que $\infty > D[v] > d_v$. Soit v le premier sommet vérifiant cette condition rajouté à la file au cours de l'exécution de l'algorithme. Nécessairement, $v \notin W$, car sinon on aurait $d_v = D[v] = 0$. On en déduit que v a été rajouté à la file au moment du traitement d'un sommet v'. Par hypothèse, on a $d_{v'} = D[v']$. Mais alors, on pose $D[v] = D[v'] + 1 = d_{v'} + 1$. Sachant que v' et v sont voisins, on en déduit qu'il existe un chemin de taille $\leq d_{v'} + 1$ menant de W à v. On en déduit $d_v \leq d_{v'} + 1 \leq D[v'] + 1 = D[v]$. On conclut par l'absurde.

$P[v]$ représente l'arête qui précède v dans un plus court chemin de W à v.

On rappelle l'existence des fonctions suivantes dans le module `Queue` qui implémente la structure abstraite de file :

`create : unit -> 'a Queue.t`. Crée une file vide.

`is_empty : 'a Queue.t`. Teste si une file est vide.

`push : 'a -> 'a Queue.t -> unit`. Ajoute un élément dans une file.

`pop : 'a Queue.t -> 'a`. Retire et renvoie l'élément de tête d'une file.

☐ 26 – Écrire une fonction `accessibles` de signature :

```
graphe -> int list -> int * int array * (int * int) array
```

prenant en entrée un graphe d'automate G et un ensemble X de sommets (sous forme de liste d'états) et qui renvoie le triplet (c, D, P) calculé selon l'algorithme précédent. Les constantes ∞, *vide* et *rien* seront respectivement codées dans cette fonction par `-1`, `(-2, -1)` et `(-1, -1)`.

Réponse 26.

```
let accessibles g x =
    let n = Array.length g in
    let f = Queue.create () in
    let d = Array.make n (-1) in
    let p = Array.make n (-2, -1) in
    let c = ref n in
    let init s =
        Queue.push f s;
        d.(s) <- 0;
        p.(s) <- (-1, -1);
        decr c
    in
    List.iter init x;
    let traiter_arete s (t, a) =
        if d.(t) = -1 then
          begin
            Queue.push f t;
            d.(t) <- d.(s) + 1;
            p.(t) <- (s, a);
            decr c
          end
    in
    while not (Queue.is_empty f) do
      let s = Queue.pop f in
      List.iter (traiter_ arete s) g.(s)
    done;
    !c, d, p
```

☐ 27 – Écrire une fonction `chemin : int -> (int * int) array -> int list` qui prend en entrée un sommet s et le tableau P calculé à l'aide de la fonction `accessibles` sur un graphe G et un ensemble X, et renvoie un mot de longueur minimale qui est l'étiquette d'un chemin d'un sommet de X à s (ou un message d'erreur s'il n'en existe pas). Un mot sera représenté par la liste de ses lettres.

Réponse 27. La fonction auxiliaire prend en argument le mot en cours de construction (on a besoin d'un accumulateur, car on veut remplir le mot par la fin) ainsi qu'un sommet et renvoie le mot nécessaire pour atteindre ce sommet depuis W. Les cas de base sont ceux où $P[v]$ vaut *vide* ou *rien*.

```
let chemin v p =
  let rec aux acc v = match p.(v) with
    | -2, -1 -> failwith "Non accessible"
    | -1, -1 -> acc
    | vv, a  -> aux (a :: acc) vv in
  aux [] v
```

Partie IV. Réduction depuis SAT – 15 pts

On cherche à étudier le problème de décision suivant :

Mot synchronisant :
Instance : une machine M et un entier m, donné en écriture unaire [a].
Solution : V si M possède un mot synchronisant de taille $\leq m$, F sinon.

a. L'intérêt de cette écriture ici est que la taille de l'entrée est égale à la somme de la taille de M et de la taille de m, or, la taille de m vaut m en unaire et $O(m)$ dans toute autre base. Un algorithme dont la complexité est $O(m)$ est donc polynomial comme m est donné en unaire alors qu'il ne le serait pas sinon

☐ 28 – Montrer que **Mot synchronisant**$\in$ NP.

Réponse 28. Si une machine M possède un mot synchronisant u tel que $|u| \leq m$, alors u est un certificat. La vérification que u est un mot synchronisant se fait en temps polynomial (cf. fonction `synchronisant` en partie I). La taille de u est $O(m)$ donc polynomial en la taille de l'instance.

On s'intéresse maintenant au problème `CNF-SAT` de satisfiabilité d'une formule logique sous forme normale conjonctive. Par exemple,

$$\varphi_0 = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee x_4)$$

est une formule sous forme normale conjonctive formée de trois clauses et portant sur un ensemble de 4 variables $\mathcal{V}_0 = \{x_1, x_2, x_3, x_4\}$.
Soit φ une formule sous forme normale conjonctive, composée de n clauses et faisant intervenir m variables d'un ensemble $\mathcal{V}$. On suppose les clauses numérotées c_1, c_2, …, 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 inférieure ou égale à m sur une certaine machine.
On introduit pour cela la machine $M_\varphi = (Q, \Sigma, \delta)$ associée à φ par :

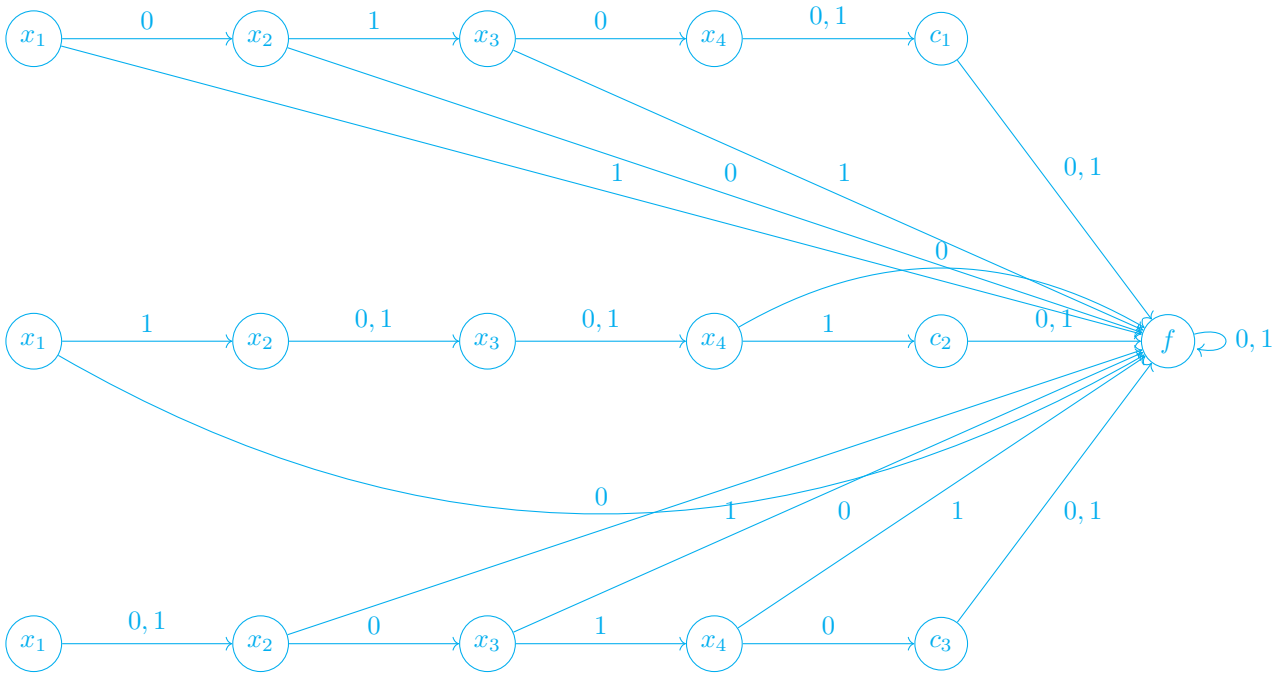
- Q est formé de $mn+n+1$ états, à savoir un état particulier noté f et $n(m+1)$ autres états qu'on notera $q_{i,j}$ avec $(i,j) \in [\![1,n]\!] \times [\![1,m+1]\!]$;
- $\Sigma = \{0,1\}$;
- δ est définie par :
  - f est un états puits, c'est-à-dire que $\delta(f,0) = \delta(f,1) = f$;
  - pour tout $i \in [\![1,n]\!]$, $\delta(q_{i,m+1},0) = \delta(q_{i,m+1},1) = f$;
  - pour tout $i \in [\![1,n]\!]$ et $j \in [\![1,m]\!]$,

$$\delta(q_{i,j},0) = \begin{cases} f & \text{si le littéral } \overline{x_j} \text{ apparaît dans la clause } c_i \\ q_{i,j+1} & \text{sinon} \end{cases}$$

$$\delta(q_{i,j},1) = \begin{cases} f & \text{si le littéral } x_j \text{ apparaît dans la clause } c_i \\ q_{i,j+1} & \text{sinon} \end{cases}$$

☐ 29 – Représenter graphiquement M_{φ_0} où φ_0 est la formule donnée précédemment.

Réponse 29.



☐ 30 – Donner un modèle $\mu \in \{F, V\}^{\mathcal{V}_0}$ de φ_0. En déduire un mot synchronisant pour M_{φ_0} de longueur 4.

Réponse 30. $\mu : x_1 \mapsto V, x_2 \mapsto V, x_3 \mapsto V, x_4 \mapsto F$.
1110 est un mot synchronisant.

☐ 31 – Montrer que tout mot u de longueur $m+1$ est synchronisant. À quelle condition sur les $\delta^*(q_{i,1}, u)$ un mot u de longueur m est-il synchronisant ?

Réponse 31. $\forall q \in Q, \forall u, |u| \geq m+1, \delta^*(q, u) = f$. On peut remarquer que si on est dans un état $q_{i,j}$, soit on se retrouve dans f, soit dans $q_{i,j+1}$. Pour un mot u de taille m, il faut que $\delta^*(q_{i,1}, u) = f$ pour que u soit synchronisant car f est un puits.

☐ 32 – Montrer que si une formule φ est satisfiable, tout modèle de φ donne un mot de longueur m synchronisant pour M_φ. On détaillera la construction de ce mot.

Réponse 32. Soit μ un modèle de φ. On pose $u = u_1 \dots u_m$, avec $u_j = 1$ si $\mu(x_j) = V$ et $u_j = 0$ si $\mu(x_j) = F$. On a alors $\delta^*(q_{i,1}, u) = f$ pour tout i. Et pour les autres états, tout chemin de longueur m mène à f.

☐ 33 – Réciproquement, montrer que si M_φ possède un mot synchronisant de longueur inférieure ou égale à m, alors φ est satisfiable. Décrire comment construire un modèle de φ.

Réponse 33. Soit u un mot synchronisant de longueur m pour M_φ.
$\delta^*(q_{i,1}, u) = f$ si et seulement si il existe j tel que $\delta(q_{i,j}, u_j) = f$ (sinon on serait dans $q_{i,m+1}$). On a alors $\forall i, \exists j$, $u_j = 0$ et $\overline{x_j}$ apparaît dans c_i ou $u_j = 1$ et x_j apparaît dans c_i. En prenant la valuation $\mu : x_j \mapsto V$ si $u_j = 1$ et F si $u_j = 0$, on obtient un modèle de φ.
Notons que si il existe un mot synchronisant de longueur $< m$, on peut le compléter en un mot synchronisant de longueur m en ajoutant des lettres.

☐ 34 – Conclure quand à la nature du problème **Mot synchronisant**.

Réponse 34. `CNF-SAT` est NP-difficile et `CNF-SAT` $\leq_P$ **Mot synchronisant** donc **Mot synchronisant** est NP-difficile. Or, il est dans la classe NP. Donc **Mot synchronisant** est NP-complet.

Partie V. Existence – 20 pts

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

☐ 35 – Soit u un mot synchronisant de M et u_0, u_1, …, 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 $|\delta^*(Q, u_i)|$?

Réponse 35. On a nécessairement $|\delta^*(Q, u_0)| \geq |\delta^*(Q, u_1)| \geq \ldots \geq |\delta^*(Q, u_r)| = 1$. 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, $|\delta^*(Q, u_r)| = 1$.

☐ 36 – 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 $\delta^*(q, u_{q,q'}) = \delta^*(q', u_{q,q'})$.

Réponse 36. 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 et q' de Q_i et un mot $u_{i+1} = u_{q,q'}$ tel que $|Q_{i+1}| = |\delta^*(Q_i, u_{i+1})| < |Q_i|$. On pose alors $v_{i+1} = v_i u_{i+1}$.

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

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 :
- $\widetilde{Q} = \mathcal{P}_1(Q) \cup \mathcal{P}_2(Q)$ est formé des parties à un ou deux éléments de Q ;
- $\widetilde{\delta}$ est définie par $\forall (X, a) \in \widetilde{Q} \times \Sigma, \widetilde{\delta}(X, a) = \{\delta(q, a), q \in X\}$.

☐ 37 – Si $n = |Q|$, que vaut $\widetilde{n} = |\widetilde{Q}|$?

Réponse 37. Il y a $\widetilde{n} = \binom{n}{1} + \binom{n}{2} = n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2}$ parties à 1 ou 2 éléments de Q.

On a dit que pour la modélisation informatique, l'ensemble d'états d'une machine doit être modélisé par un intervalle $[\![0, n-1]\!]$. $\widetilde{Q}$ doit donc être modélisé par $[\![0, \widetilde{n}-1]\!]$. On pose φ_n une bijection de $\widetilde{Q}$ sur $[\![0, \widetilde{n}-1]\!]$.

☐ 38 – En détaillant la bijection φ_n choisie, écrire une fonction `phi : int -> int list -> int` qui prend en argument l'entier n correspondant au nombre d'état, ainsi qu'une liste à un élément ou deux éléments distincts ℓ et renvoie $\varphi_n(\{q\})$ si $\ell = [q]$ et $\varphi_n(\{q_1, q_2\})$ si $\ell = [q_1; q_2]$ avec $q_1 \neq q_2$.

☐ 39 – Après avoir montré 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 une bijection de $\widetilde{Q}$ sur $[\![0, \widetilde{n}-1]\!]$, écrire une fonction `phi : int -> int list -> int` qui prend en argument l'entier n correspondant au nombre d'état, ainsi qu'une liste à un élément ou deux éléments distincts ℓ et renvoie $\varphi_n(\{q\})$ si $\ell = [q]$ et $\varphi_n(\{q_1, q_2\})$ si $\ell = [q_1; q_2]$ avec $q_1 \neq q_2$.

Réponse 39. On remarque que $\{[\![\frac{q(q+1)}{2}, \frac{q(q+1)}{2} + q]\!], q \in [\![0, n-1]\!]\}$ forme une partition de $[\![0, \widetilde{n}-1]\!]$. 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.

```
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 [q1]
    | [q1 ; q2]              -> phi n [q2 ; q1]
```

On admettra disposer de sa fonction réciproque `phi_inv : int -> int -> int list`.

Pour la suite, on représente une machine $M = (Q, \Sigma, \delta)$ par la seule donnée d'une matrice (tableau de tableau) d'entiers `m` telle que `m.(q).(a)` représente l'état $\delta(q, a)$. Par exemple, la machine M_0 représentée graphiquement sur la figure 1 a pour implémentation OCaml :

```
type machine = int array array;;

let m0 = [| [|1; 1|]; [|0; 2|]; [|0; 2|] |]
```

☐ 40 – Écrire une fonction `delta_tilde : machine -> int -> lettre -> int` qui prend en argument une machine M, $X \in \widetilde{Q}$ et $a \in \Sigma$ et renvoie $\widetilde{\delta}(X, a)$.

Réponse 40. On distingue selon que W soit ou non un singleton. Si W est une paire, on n'a pas à distinguer selon que les deux images par la lecture de la lettre soient égales ou non, car la fonction précédente gère déjà ce cas.

```
let delta_tilde m q a =
    let n = nombre_etats m in match phi_inv n q with
        | [x]    -> phi n [m.(x).(a)]
        | [x; y] -> phi n [m.(x).(a); m.(y).(a)]
```

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 $\{(X, a, \widetilde{\delta}(X, a)), (X, a) \in \widetilde{Q} \times \Sigma\}$. On associe alors à $\widetilde{G}$ le graphe transposé $\widetilde{G}^T$ qui a les mêmes sommets que $\widetilde{G}$ mais dont les arcs sont retournés (i.e. (X, a, Y) est un arc de $\widetilde{G}^T$ si et seulement si (Y, a, X) est un arc de $\widetilde{G}$).

☐ 41 – Écrire une fonction `transpose : machine -> (int * int) list array` qui à partir d'une machine M calcule le tableau des listes d'adjacences de $\widetilde{G}^T$.

Réponse 41. On parcourt toutes les lettres et tous les états pour déterminer quelles transitions mènent à un état donné, afin de mettre à jour sa liste d'adjacence.

```
let transpose m =
    let n = Array.length m and p = Array.length m.(0) in
    let n_tilde = n * (n + 1) / 2 in
    let gr = Array.make n_tilde [] in
    for q = 0 to n_tilde - 1 do
        for a = 0 to p - 1 do
            let qq = delta_tilde m q a in
            gr.(qq) <- (q, a) :: gr.(qq)
        done
    done;
    gr
```

☐ 42 – Justifier qu'il suffit d'appliquer la fonction `accessibles` de la partie au graphe $\widetilde{G}^T$ et à l'ensemble des sommets de $\widetilde{G}^T$ correspondant à des singletons pour déterminer si la machine M possède un mot synchronisant.

Réponse 42. Le résultat de la question 36 peut se reformuler en disant qu'il existe toujours une suite de transitions dans $\widetilde{M}$ d'un état qui est une paire vers un état qui est singleton. Cela signifie que dans $\widetilde{G_R}$, tous les états sont accessibles depuis les singletons. On peut donc appliquer la fonction `accessibles` à l'ensemble formé des singletons pour savoir s'il existe un mot synchronisant.

☐ 43 – Écrire une fonction `existe_synchronisant : machine -> bool` qui détermine si une machine possède un mot synchronisant.

Réponse 43. On utilise les fonctions précédentes. On vérifie qu'après l'exécution, on a $c = 0$. Ici, la fonction auxiliaire sert à créer la liste des singletons.

```
let existe_synchronisant m =
    let gr = transpose m and n = Array.length m in
    let rec aux = function
        | 0 -> []
        | k -> (phi n [k - 1]) :: aux (k - 1) in
    let c, _, _ = accessibles gr (aux n) in
    c = 0
```

Partie VI. Jeux sur machine non déterministe – 10 pts

On s'intéresse maintenant à des machines non-déterministes. Une machine non-déterministe est une machine dont la fonction de transition δ est une application de $Q \times \Sigma$ dans $\mathcal{P}(Q)$.
La fonction de transition étendue devient :

- pour tout $q \in Q$, $\delta^*(q, \varepsilon) = \{q\}$;
- pour tous $q \in Q$, $u \in \Sigma^*$ et $a \in \Sigma$, $\delta^*(q, ua) = \cup_{q' \in \delta^*(q,u)} \delta(q', a)$.

Pour deux états q et q', q' est dit accessible depuis q s'il existe un mot $u \in \Sigma^*$ tel que $q' \in \delta^*(q, u)$.
Un mot $u \in \Sigma^*$ est synchronisant pour (Q, σ, δ) s'il existe $q_0 \in Q$ tel que $\forall q \in Q$, $\delta^*(q, u) = \{q_0\}$, c'est-à-dire que tous les chemins partant de tout état mènent à q_0 en lisant u.
On définit le jeu d'accessibilité suivant à 2 joueurs sur une machine non-déterministe M.

Arène : une machine non-déterministe $M = (\Sigma, Q, \delta)$, ainsi qu'un ensemble d'états gagnants pour J1 $F \subset Q$.

Initialisation : au début de la partie, J2 choisit un état $q \in Q$.

Tour de J1 : à chacun de ses tours, J1 choisit une lettre $a \in \Sigma$.

Tour de J2 : suite au choix de a par J1, J2 choisit le nouvel état $q' \in \delta(q, a)$, qui remplace q.

Fin de la partie : si à la fin du tour de J2, $q \in F$, alors la partie s'arrête et J1 gagne. Si la partie est infinie, alors J2 gagne.

☐ 44 – Montrer que si $F \neq Q$ et que $\forall q \in Q, \forall a \in \Sigma, \delta(q, a) = Q$, alors J2 possède une stratégie gagnante.

Réponse 44. Comme $F \neq Q$, il existe un état $q \notin F$. J2 choisit cet état. Ensuite, quelle que soit la lettre a choisie par J1, $q \in \delta(q, a)$, donc J2 peut choisir de rester dans l'état q.

☐ 45 – On suppose que M possède un mot synchronisant, et que l'ensemble d'états gagnants F est un singleton $\{q_f\}$. Montrer que si J1 peut choisir q_f, alors il a une stratégie gagnante.

Réponse 45. Si M possède un mot synchronisant u, alors il existe un état q_0 tel que $\forall q, \delta^*(q, u) = \{q_0\}$. C'est-à-dire que quels que soient les choix de J2, si J1 choisit successivement les lettres du mot u, on arrivera dans l'état q_0. Donc si J1 peut choisir l'ensemble d'états gagnants $\{q_0\}$, alors il possède une stratégie gagnante.

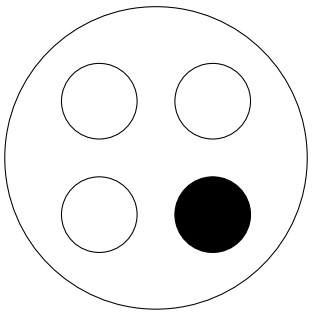
On s'intéresse maintenant au problème suivant du barman aveugle avec des gants de boxe :

Un barman aveugle avec des gants de boxes doit mettre 4 verres disposés sur un plateau dans le même sens : tous à l'endroit ou tous à l'envers.

Les règles sont les suivantes :

- Quatre verres sont sur un plateau. Ils sont disposés aux sommets d'un carré, tantôt à l'endroit tantôt à l'envers.
- Un barman aveugle avec des gants de boxe essaie de mettre tous les verres dans le même sens (peu importe le sens).
- Avant chaque essai, si tous les verres sont dans le même sens, le barman est averti, sinon une personne tourne le plateau à sa guise, puis, le barman réalise son essai.

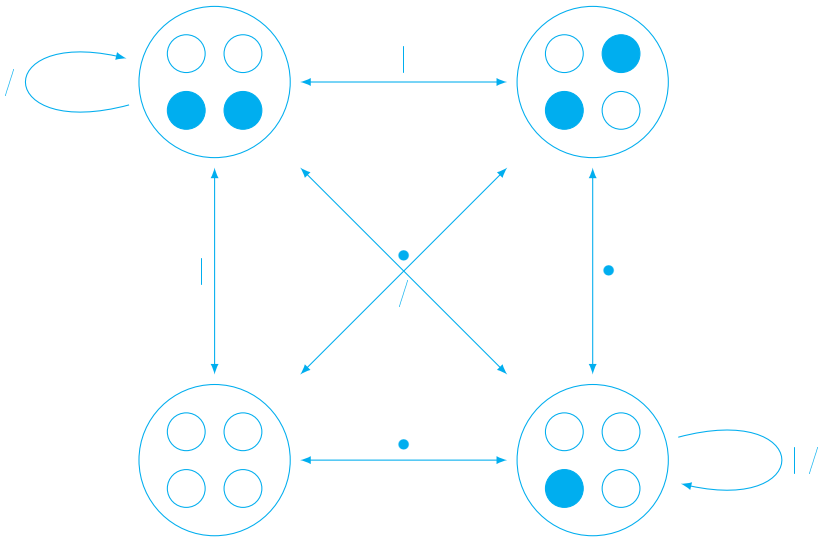
Voici un exemple de configuration possible :



☐ 46 – Dessiner un automate modélisant le problème comme une instance du jeu d'accessibilité (on pourra le réduire à 4 états en tenant compte des symétries du problème).

☐ 47 – Dessiner un automate modélisant le problème comme une instance du jeu d'accessibilité (on pourra tenir compte des symétries du problème pour réduire le nombre d'états).

Réponse 47. Si on note '•' l'action de retourner 1 verre, '|' l'action de retourner 2 verres sur un même côté du carré, et '/' l'action de retourner 2 verres en diagonales, en remarquant que 1 verre retourné et 3 verres retournés sont des états équivalents, on se ramène au jeu sur la machine suivante :



☐ 48 – Donner un mot synchronisant pour cet automate ou justifier qu'il n'en existe pas.

Réponse 48. Si on s'intéresse à la parité du nombre de verres retournés. Les actions '/' et '|' ne changent pas cette parité, et l'action • la change. En lisant un même mot en partant de l'état dans lequel le nombre de verres retournés est impair, on ne tombera jamais dans le même état que si on était parti d'un état avec un nombre de verres retournés pair. Il n'existe donc pas de mot synchronisant pour cette machine.

☐ 49 – Donner une stratégie gagnante pour le barman.

Réponse 49. /|/•/|/

Fin de l'épreuve