COMPOSITION D'INFORMATIQUE n°3

Corrigé

⋆ ⋆ ⋆

1 Présentation du jeu

Question 1 On montre ce résultat par récurrence :

- $q_0 = \delta^*(q_0, \varepsilon) = \delta^*(q_0, u_0)$;
- supposons que pour un certain $k \geqslant 0$, $q_k = \delta^*(q_0, u_k)$. Alors :

$$q_{k+1} = \delta(q_k, a_{k+1}) = \delta(\delta^*(q_0, u_k), a_{k+1}) = \delta^*(q_0, u_k a_{k+1}) = \delta^*(q_0, u_{k+1})$$

On conclut par récurrence.

Question 2 Si $\Sigma = \{a\}$, alors toutes les stratégies sont égales (et renvoient toujours a). Deux cas sont alors possibles :

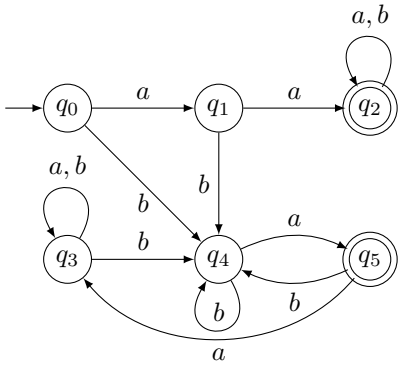
- si $L = \emptyset$, aucune partie ne peut être gagnée par 1, donc le joueur 2 possède une stratégie gagnante ;
- sinon, soit $u \in L$ de taille minimale. Alors $u = a^k$. Toute partie sera gagnée par le joueur 1 après k coups, donc le joueur 1 possède une stratégie gagnante.

Question 3 Il suffit de rajouter un nouvel état initial, dont les transitions sortantes pointent toutes vers l'ancien état initial. Cela change le langage reconnu par l'automate, mais, mis à part ce nouvel état, les attracteurs des deux joueurs ne seront pas modifiés.

Formellement, on pose $A' = (Q \cup \{q_0'\}, \delta', q_0', F)$ où :

- pour $q \in Q$, $a \in \Sigma$, $\delta'(q, a) = \delta(q, a)$;
- pour $a \in \Sigma$, $\delta'(q_0', a) = q_0$.

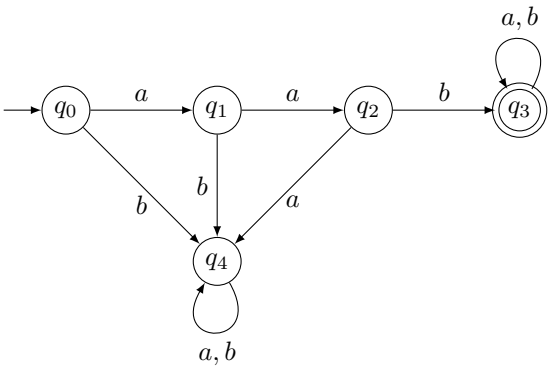
Question 4 On propose :



On propose la stratégie suivante pour le joueur 1 : $f(q_0) = f(q_1) = f(q_4) = a$ et les autres valeurs sont arbitraires. Cette stratégie est gagnante :

- si $g(q_1) = a$, alors la (f, g)-partie sera aa (si 1 commence ou si 2 commence et $g(q_0) = a$) ou ba (si 2 commence et $g(q_0) = b$) gagnantes pour 1 ;
- si $g(q_1) = b$, alors la (f, g)-partie sera aba (si 1 commence), aa (si 2 commence et $g(q_0) = a$) ou ba (si 2 commence et $g(q_0) = b$), gagnantes pour 1.

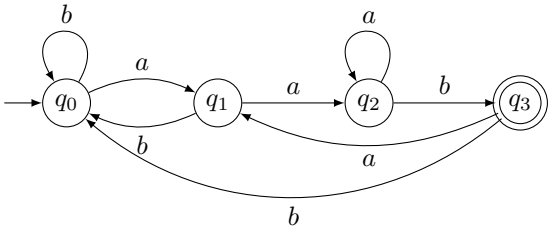
Question 5 On propose :



On propose la stratégie suivante pour le joueur 2 : $g(q_1) = b$ et les autres valeurs sont arbitraires. Cette stratégie est gagnante :

- si $f(q_0) = a$, alors la (f, g)-partie sera ab, gagnante pour 2 car q_4 n'est pas co-accessible ;
- si $f(q_0) = b$, alors la (f, g)-partie sera b, gagnante pour 2 car q_4 n'est pas co-accessible.

Question 6 On propose :



L'automate n'ayant pas d'état non co-accessible, le joueur 2 n'a pas de stratégie gagnante. Montrons que le joueur 1 n'a pas de stratégie gagnante. Par l'absurde, supposons f une stratégie gagnante pour 1. On pose : $g(q_0) = b$, $g(q_1) = b$ et les autres valeurs choisies arbitrairement. Alors la (f, g)-partie n'est pas gagnée par le joueur 1. En effet :

- si $f(q_0) = b$, alors la (f, g)-partie est la suite ne contenant que des b, qui stationne en q_0 ;
- si $f(q_0) = a$, alors la (f, g)-partie est la suite qui alterne des a et des b, alternant entre q_1 et q_0.

C'est absurde. Il n'existe donc pas de stratégie gagnante pour le joueur 1.

2 Jeu dans un langage fini

Question 7 Il suffit de montrer qu'il n'existe pas de partie nulle. Supposons $(a_i)_{i \in \mathbb{N}^*}$ une partie. On distingue :

- s'il existe $i \in \mathbb{N}^*$ tel que $u_i \in L$, alors la partie est gagnée par le joueur 1 après i coups ;
- sinon, soit m la longueur maximale d'un mot de L. Dans un automate reconnaissant L, l'état q_{m+1} est nécessairement un état non co-accessible, donc la partie est gagnée par le joueur 2 après $m + 1$ coups.

Question 8 On se contente de calculer les images par la fonction de transition. L'automate étant complet, il n'y a pas à vérifier que les transitions existent.

```
int delta_etoile(afd A, int q, char* u){
    for (int k=0; u[k] != '\0'; k++){
        q = A.delta[q][phi(u[k])];
    }
    return q;
}
```

Question 9 Il suffit de vérifier si $\delta^*(q_0, u) \in F$.

```
bool reconnu(afd A, char* u){
    int q = delta_etoile(A, 0, u);
    return A.finaux[q];
}
```

Question 10 S'il existe des états non accessibles, les supprimer de l'automate ne change pas le langage reconnu, et laisse l'automate complet, ce qui contredit la minimalité du nombre d'état.

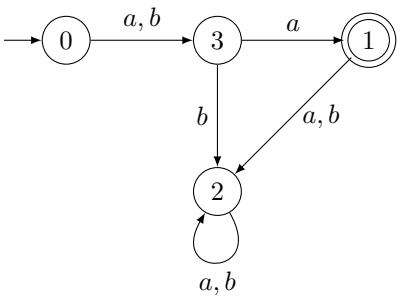
Si A ne possède pas d'état non co-accessible, alors le langage L est infini. En effet, soit $a \in \Sigma$. L'état $\delta^*(q_0, a^n)$ est co-accessible pour tout $n \in \mathbb{N}$, donc il existe u_n tel que $a^n u_n \in L$, ce qui montre bien le caractère infini, qui est absurde.

Si A possède au moins deux états non co-accessibles, on peut les « fusionner » sans changer le langage reconnu ou le caractère complet, ce qui contredit la minimalité du nombre d'état.

Question 11 Supposons qu'il existe $u \in \Sigma^+$ tel que $\delta^*(q, u) = q$. Si q est co-accessible, il existe $v \in \Sigma^*$ tel que $\delta^*(q, v) \in F$. Dans ce cas $\{u^n v \mid n \in \mathbb{N}\} \subseteq L$. Or le langage $\{u^n v \mid n \in \mathbb{N}\}$ est infini, ce qui contredit le caractère fini de L. On en déduit que q n'est pas co-accessible, donc $q = q_\omega$.

Réciproquement, l'automate A étant complet, pour $a \in \Sigma$, $\delta(q_\omega, a)$ est bien défini. Si on suppose $\delta(q_\omega, a) = q' \neq q_\omega$, alors q' est co-accessible, donc q_ω serait co-accessible, ce qui est absurde. On en déduit que $\delta(q_\omega, a) = q_\omega$, et $a \in \Sigma^+$.

Question 12 On propose l'automate suivant. On fait attention aux valeurs des états pour que ça corresponde bien au tableau donné.



Question 13

(a) Si q possède un voisin d'issue 0, le joueur dont c'est le tour peut choisir une lettre qui amène vers cet état, donc c'est le premier joueur qui a une stratégie gagnante, ce qui contredit l'hypothèse que l'issue est 0. Si q possède un voisin d'issue 1, alors si c'est au tour du joueur 1 de jouer, il peut choisir une lettre qui amène vers cet état d'issue 1, donc il possède une stratégie gagnante depuis q. De même si q possède un voisin d'issue 2.

Réciproquement, si c'est au joueur 1 de jouer, il n'y a aucun voisin d'issue 0 ou 1, donc c'est le joueur 2 qui a une stratégie gagnante. De même symétriquement. Dans les deux cas, c'est le joueur dont ce n'est pas le tour qui a une stratégie gagnante.

(b) Si q possède un voisin d'issue 0 ou d'issue 2, alors le joueur 2 aurait une stratégie gagnante si c'est son tour de jouer. Si q ne possède pas de voisin d'issue 1, tous les voisins seraient d'issue 3, donc l'issue de q serait 0, pas 1.

Réciproquement, si c'est au joueur 1 de jouer, il choisit un voisin d'issue 1, si c'est au joueur 2 de jouer, tous les voisins sont d'issue 1 ou 3. Dans les deux cas, c'est bien le joueur 1 qui a une stratégie gagnante.

(c) On raisonne de même symétriquement.

(d) Ce dernier cas correspond à tous les cas non traités précédemment (et l'issue d'un état correspond bien à l'une des quatre situations décrites).

Question 14 On commence par écrire la fonction suggérée. On suppose ici que `issue[q]` vaut -1 si l'état q n'a pas encore été exploré.

Si l'état a déjà été exploré, on renvoie la valeur de son issue. Sinon, on commence par déterminer s'il s'agit d'un état final ou l'état puits. Dans ce cas, on ne renvoie pas tout de suite l'issue, car il faut explorer les voisins. On utilise un tableau `voisins` qui garde en mémoire le nombre d'occurrences de chaque issue parmi les voisins de q. On lance un appel récursif depuis chaque voisin et on modifie le tableau `voisins` en conséquence. Une fois le calcul terminé, on détermine l'issue de q selon la question précédente.

```
int issue_rec(afd A, int q, int* issue){
    if (issue[q] != -1) return issue[q];
    if (A.finaux[q]) issue[q] = 1;
    if (A.delta[q][0] == q) issue[q] = 2;
    int voisins[4] = {0, 0, 0, 0};
    for (int a=0; a<A.Sigma; a++){
        int p = A.delta[q][a];
        voisins[issue_rec(A, p, issue)]++;
    }
    if (issue[q] != -1){
        if (voisins[0] + voisins[1] * voisins[2] > 0) issue[q] = 3;
        else if (voisins[1] > 0) issue[q] = 1;
        else if (voisins[2] > 0) issue[q] = 2;
        else issue[q] = 0;
    }
    return issue[q];
}
```

Dès lors la fonction demandée consiste uniquement à créer le tableau `issue` et à lancer un appel depuis l'état 0 (qui permet d'atteindre tous les états).

```
int* calcul_issues(afd A){
    int* issue = malloc(A.Q * sizeof(int));
    for (int q=0; q<A.Q; q++) issue[q] = -1;
    issue_rec(A, 0, issue);
    return issue;
}
```

Le principe de la fonction `issue_rec` en font un parcours de graphe. À part les appels récursifs, les autres opérations se font en temps constant. La complexité totale est donc linéaire en le nombre d'arêtes parcourues, c'est-à-dire $\mathcal{O}(|Q||\Sigma|)$ (car l'automate est déterministe complet).

La fonction `calcul_issues` crée le tableau `issue` (en temps $\mathcal{O}(|Q|)$) puis lance un appel à `issue_rec`, ce qui donne bien la complexité attendue.

Question 15 Le joueur j a une stratégie gagnante depuis q lorsque c'est son tour si et seulement si $\iota(q) \in \{j, 3\}$. Si $\iota(q) = j$, il possède un voisin d'issue j ; si $\iota(q) = j$, il possède un voisin d'issue 0 ou j. On cherche le premier voisin vérifiant cette condition et on renvoie la lettre associée. Si on n'a trouvé aucun voisin correspondant à la condition, on renvoie la lettre de valeur 0.

```
char strategie_optimale(afd A, int q, int j, int* issue){
    for (int a = 0; a<A.Sigma; a++){
        int iss = issue[A.delta[q][a]];
        if (iss == 0 || iss == j) return psi(a);
    }
    return psi(0);
}
```

3 Langages continuables et mots primitifs

Question 16 Σ^* est un langage continuable (trivialement). Pour $a \in \Sigma$, $a\Sigma^*$ est un langage rationnel infini non continuable (tout mot ne commençant pas par a ne peut pas être continué en un mot du langage).

Question 17 Pour $a \in \Sigma$, $\Sigma^* a$ est un langage rationnel et continuable. Son complémentaire est l'ensemble des mots ne terminant pas par a et est bien infini.

Question 18 ($\Rightarrow$) Soit L un langage continuable (donc rationnel par hypothèse) et $A = (Q, \delta, q_0, F)$ un automate fini déterministe émondé le reconnaissant (un tel automate existe car $L \neq \emptyset$, qui n'est pas continuable). Supposons que A n'est pas complet et soit $(q, a) \in Q \times \Sigma$ un blocage. L'état q étant accessible, il existe $u \in \Sigma^*$ tel que $\delta^*(q_0, u) = q$. Dès lors, pour tout mot $v \in \Sigma^*$, $uav \notin L$, car la lecture de ua dans l'automate ne peut pas aboutir. On en déduit que L n'est pas continuable, ce qui est absurde.

($\Leftarrow$) Soit $A = (Q, \delta, q_0, F)$ un automate fini déterministe complet émondé. Pour $u \in \Sigma^*$, l'état $q = \delta^*(q_0, u)$ est bien défini (car A est complet) et co-accessible (car A est émondé). Il existe donc $v \in \Sigma^*$ tel que $\delta^*(q, v) \in F$. On en déduit que $uv \in L$, donc L est continuable.

Question 19 $abaaabaa = (abaa)^2$ n'est pas primitif.

Un mot de longueur 17 est primitif car si $u = v^p$ avec $v \neq \varepsilon$ et $p > 1$, $|u| = p \times |v|$ n'est pas un nombre premier (contrairement à 17).

Question 20 Si $u = a_1 \dots a_n$, pour chaque entier $k \in [\![1, \frac{n}{2}]\!]$, on vérifie si k divise n et si $u = (a_1 \dots a_k)^{\frac{n}{k}}$.

```
bool primitif(char* u){
    int n = strlen(u);
    for (int k=1; k<=n/2; k++){
        if (n % k == 0){
            int i = k;
            while (i < n && u[i] == u[i % k]){
                i++;
            }
            if (i == n) return false;
        }
    }
    return true;
}
```

La fonction n'utilise pas d'espace mémoire autre que les variables n et k, le calcul de `strlen(u)` est en $\mathcal{O}(n)$ et l'enchaînement des deux boucles en $\mathcal{O}(n^2)$.

Question 21 Le langage aa^+ est infini et ne contient que des mots non primitifs. Le langage a^*b est infini et ne contient que des mots primitifs.

Question 22 Soit A un automate fini à n états dont $L(A)$ est continuable. On peut supposer que tous les états de A sont accessibles et co-accessibles par la question 18. Dès lors, $\forall u \in \Sigma^*, \exists v \in \Sigma^*$ tel que $|v| < n$ et $uv \in L(A)$.

On pose alors, pour $i \in \mathbb{N}$, $i \geqslant n-1$, $u_i = ab^i$. On pose de plus v_i le plus petit mot tel que $w_i = u_i v_i \in L(A)$.

On a $|v_i| < n$, donc w_i est un mot primitif. En effet, si $w_i = x^p$ avec $p \geqslant 2$, alors x (la première occurrence) commence par un a (car u_i commence par un a), mais comme $|u_i| \geqslant n > |v_i|$, on a $|x| < |u_i|$, et donc x (la deuxième occurrence) commence par un b.

C'est absurde, donc w_i est un mot primitif. Il existe bien une infinité de tels mots (car $1+i+n > |w_i| > i$).

Question 23 La réciproque est fausse, par exemple avec a^*b (tous les mots sont primitifs, mais il n'est pas continuable).

4 Jeu dans un langage non continuable

4.1 Manipulation de listes

Question 24 On fait un `malloc` et on attribue les champs.

```
liste* cons(int x, liste* lst){
    liste* res = malloc(sizeof(liste));
    res->x = x;
    res->suivant = lst;
    return res;
}
```

Question 25 On parcourt la liste (ici récursivement, mais une boucle conviendrait aussi).

```
bool mem(int x, liste* lst){
    if (lst == NULL) return false;
    return lst->x == x || mem(x, lst->suivant);
}
```

4.2 Graphes d'automates

Question 26 On commence par créer un tableau de listes vides. Ensuite, on parcourt chaque transition de l'automate, et si le voisin n'est pas déjà dans la liste d'adjacence, on le rajoute. La complexité est en $\mathcal{O}(|Q||\Sigma|^2)$ (on pourrait avoir $\mathcal{O}(|Q||\Sigma|)$ avec des structures plus efficaces).

```
graphe graphe_automate(afd A){
    liste** T = malloc(A.Q * sizeof(liste*));
    for (int q=0; q<A.Q; q++) T[q] = NULL;
    for (int q=0; q<A.Q; q++){
        for (int a=0; a<A.Sigma; a++){
            int p = A.delta[q][a];
            if (!mem(p, T[q]))
                T[q] = cons(p, T[q])
        }
    }
    graphe G = {.Q = A.Q, .T = T};
    return G;
}
```

Question 27 On commence par écrire une fonction de libération de la mémoire d'une liste, qu'on appelle sur chaque liste d'adjacence.

```
void liberer_liste(liste* lst){
    if (lst != NULL){
        liberer_liste(lst->suivant);
        free(lst);
    }
}

void liberer_graphe(graphe G){
    for (int q=0; q<G.Q; q++){
        liberer_liste(G.T[q]);
    }
    free(G.T);
}
```

Question 28 L'ensemble $\mathrm{Attr}_i(X, j)$ correspond aux couples (q, ℓ) pour lesquels le joueur j a une stratégie depuis l'état q qui permet d'atteindre l'ensemble X en moins de i coups si c'est au joueur ℓ de commencer à jouer.

En effet, c'est vrai par définition pour $i = 0$, et sinon, selon que ce soit au joueur j ou $3 - j$ de jouer, on peut garantir au coup suivant d'être dans $\mathrm{Attr}_{i-1}(X, j)$.

Question 29 Par inclusion, la suite $(\mathrm{Attr}_i(X, j))_{i\in\mathbb{N}}$ est croissante. La suite des cardinaux est donc une suite d'entiers croissante et majorée par $2|Q|$. Elle est donc stationnaire au bout d'un certain rang, donc la suite $(\mathrm{Attr}_i(X, j))_{i\in\mathbb{N}}$ l'est également (par inclusion), donc convergente.

Question 30 Par les deux questions précédentes, on peut en déduire que $\mathrm{Attr}(F, 1)$ correspond aux couples (q, ℓ) tels que le joueur 1 a une stratégie gagnante depuis q si c'est à ℓ de jouer. On en déduit bien que le joueur 1 a une stratégie gagnante dans A si et seulement si $(q_0, 1) \in \mathrm{Attr}(F, 1)$ (car on a supposé que c'est le joueur 1 qui commence à jouer).

De même, le joueur 2 a une stratégie gagnante si et seulement si $(q_0, 1) \in \mathrm{Attr}(\Omega, 2)$, où Ω est l'ensemble des états non co-accessibles de A.

5 Jeu dans un langage continuable

Question 31 Le calcul ne $\mathrm{Attr}(F, 1)$ ne change pas avec ces hypothèses. Ainsi, si $(q_0, 1) \in \mathrm{Attr}(F, 1)$, le

joueur 1 a une stratégie gagnante, et sinon c'est le joueur 2 qui a une stratégie gagnante.

Question 32 On initialise un tableau de listes vides, puis on parcourt chaque liste d'adjacence pour créer une transition dans l'autre sens.

```
graphe transpose(graphe G){
    liste** Tt = malloc(G.Q * sizeof(liste*));
    for (int q=0; q<A.Q; q++) T[q] = NULL;
    for (int q=0; q<A.Q; q++){
        for (liste* lst=G.T[q]; lst!=NULL; lst=lst->suivant){
            int p = lst->x;
            Tt[p] = cons(q, Tt[p]);
        }
    }
    graphe Gt = {.Q = G.Q, .T = Tt};
    return Gt;
}
```

Question 33 Cette fonction était longue et difficile à écrire, uniquement pour occuper les plus rapides.

L'idée est de déterminer, pour chaque état $q \in Q$, si $(q, 1) \in \mathrm{Attr}(F, 1)$ et si $(q, 2) \in \mathrm{Attr}(F, 1)$.

On commence par deux petites fonctions utilitaires pour calculer le tableau des degrés d'un graphe.

```
int len(liste* lst){
    int n = 0;
    for (; lst != NULL; lst = lst->suivant) n++;
    return n;
}

int* degres(graphe G){
    int* deg = malloc(G.Q * sizeof(int));
    for (int q=0; q<G.Q; q++) deg[q] = len(G.T[q]);
    return deg;
}
```

Ensuite, on écrit une fonction récursive de parcours de graphe qui s'occupe de déterminer l'attracteur. Avec les noms de variables suivants :

- `attr[l][q]` vaut `true` si on a pu déterminer que $(q, \ell + 1) \in \mathrm{Attr}(F, 1)$;
- `deg[l][q]` indique le nombre de voisins p de q, dans G, dont on n'a pas encore déterminé que $(p, 2-\ell) \in \mathrm{Attr}(F, 1)$.

Le principe de l'algorithme est le même que celui du cours pour le calcul de l'attracteur dans un graphe biparti.

```
void dfs(graphe Gt, int* deg[2], bool* attr[2], int q, int l){
    if (attr[l][q]) return;
    attr[l][q] = true;
    for (liste* lst = Gt.T[q]; lst != NULL; lst = lst->suivant){
        int p = lst->x;
        deg[1-l][p] = deg[1-l][p] - 1;
        if (l == 1 || deg[1-l][p] == 0)
            dfs(Gt, deg, attr, p, 1 - l);
    }
}
```

Finalement, on peut écrire la fonction voulue en déterminant si $(q_0, 1) \in \mathrm{Attr}(F, 1)$.

```
int gagnant_continuable(afd A){
    graphe G = graphe_automate(A);
    graphe Gt = transpose(G);
    int* deg[2] = {NULL, NULL};
    bool* attr[2] = {NULL, NULL};
    deg[0] = degres(G);
    deg[1] = degres(G);
    attr[0] = malloc(A.Q * sizeof(bool));
    attr[1] = malloc(A.Q * sizeof(bool));
    for (int q=0; q<A.Q; q++){
        attr[0][q] = false;
        attr[1][q] = false;
    }
    for (int q=0; q<A.Q; q++){
        if (A.finaux[q]){
            dfs(Gt, deg, attr, q, 0);
            dfs(Gt, deg, attr, q, 1);
        }
    }
    bool res = attr[0][0];
    free(deg[0]); free(deg[1]);
    free(attr[0]); free(attr[1]);
    liberer_graphe(G); liberer_graphe(Gt);
    return res?1:2;
}
```

⋆⋆⋆