Devoir maison n°5

Corrigé

⋆ ⋆ ⋆

Jeux impartiaux et automates à repli

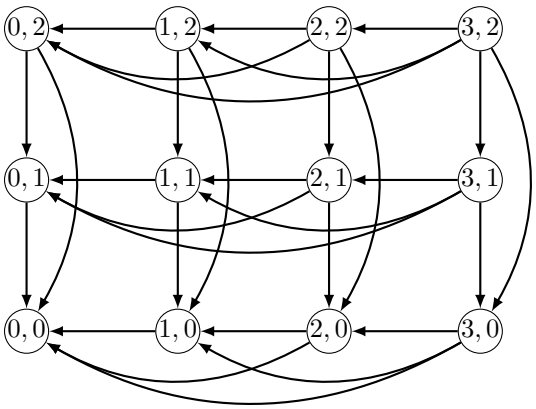
1 Étude de jeux impartiaux

1.1 Introduction

Question 1 Les jeux de Morpion, Domineering et Hex ne sont pas impartiaux : chaque joueur se différencie de l'autre (symbole pour le morpion, orientation du domino pour le domineering, côtés à relier pour le Hex).

Le Chomp et le Nim sont impartiaux. Les coups à jouer sont les mêmes, quel que soit le premier joueur.

Question 2 On obtient :



Question 3 On pose $W_{m,n} = (S, A)$ où :

- $S = [\![0, m]\!] \times [\![0, n]\!]$;
- $A = \{((p,q),(p-k,q)) \mid (p,q) \in S, k \in [\![1,p]\!]\}$
  $\cup \{((p,q),(p,q-k)) \mid (p,q) \in S, k \in [\![1,q]\!]\}$
  $\cup \{((p,q),(p-k,q-k)) \mid (p,q) \in S, k \in [\![1,\min(p,q)]\!]\}$.

Question 4 Naïvement, on teste toutes les coups possibles. Si l'un des coups mène vers un défaite du prochain premier joueur (c'est-à-dire du second joueur), le premier joueur a une stratégie gagnante. Sinon, c'est le deuxième joueur qui a une stratégie gagnante.

```
bool wythoff_naif(int m, int n){
    for (int k = 1; k <= m || k <= n; k++){
        if ((k <= m && !wythoff_naif(m - k, n)) ||
            (k <= n && !wythoff_naif(m, n - k)) ||
            (k <= m && k <= n && !wythoff_naif(m - k, n - k))){
                return true;
            }
    }
    return false;
}
```

Question 5 Supposons $m \leqslant n$. Montrons que $C(m,n) \geqslant C(0,n) = 2^n$. La première inégalité est due au fait que le premier appel récursif effectué est celui à `wythoff_naif(m - 1, n)`. Par récurrence directe, on fera donc un appel à `wythoff_naif(0, n)` avant de terminer.

Dès lors, montrons par récurrence forte sur n que $C(0,n) = 2^n$:

- $C(0,0) = 1$: il y a un seul appel, qui termine immédiatement, car la boucle est vide ;
- supposons $C(0,i) = 2^i$ pour tout $i \leqslant n \in \mathbb{N}$ fixé. Remarquons que pour $1 \leqslant k \leqslant n$, l'appel avec $(0, n+1-k)$ renvoie `true` (car la stratégie gagnante consiste à prendre toutes les pièces du deuxième tas). On en déduit que tous les appels récursifs, jusqu'au dernier (avec $(0,0)$) seront effectués. On a donc :

$$C(0,n+1) = 1 + \sum_{i=0}^{n} C(0,i) = 1 + \sum_{i=0}^{n} 2^i = 2^{n+1}$$

On conclut par récurrence.

Supposons $m > n$. Montrons que $C(m,n) \geqslant 2^m$, par récurrence forte sur $m \in \mathbb{N}^*$:

- si $m = 1$, alors $n = 0$ et $C(1,n) = 2 \geqslant 2^1$;
- supposons $C(i,n) \geqslant 2^i$ pour $i \leqslant m \in \mathbb{N}^*$ fixé, avec $i > n$. Montrons par récurrence sur $n < m+1$ que $C(m+1,n) \geqslant 2^{m+1}$:
  - si $n = 0$, comme la récurrence précédente, $C(m+1,0) = 2^{m+1}$;
  - supposons le résultat vrai pour $0 \leqslant n-1 < m$ fixé. Distinguons :
    - si (m,n) est une position perdante, alors l'appel avec $(m+1,n)$ ne fera que l'appel récursif à (m,n). Cet état étant perdant, l'appel pour (m,n) fera tous les appels récursifs à (m,i), $i < n$ et à $(m-1,i)$, $i \leqslant n$. En particulier, sachant que $m > n > 0$:

      $$C(m+1,n) \geqslant 1 + C(m,n-1) + C(m-1,n) + C(m-1,n-1) \geqslant 2^m + 2^{m-1} + 2^{m-1} = 2^{m+1}$$

    - sinon, (m,n) est une position gagnante et on a :

      $$C(m+1,n) \geqslant 1 + C(m,n) + C(m+1,n-1) \geqslant C(m+1,n-1) \geqslant 2^{m+1}$$

  On conclut par récurrence sur n.

On conclut par récurrence sur m.

Remarque

> On aurait pu faire une fonction moins efficace qui fait systématiquement tous les appels récursifs, cela aurait simplifié l'étude.

Question 6 On peut utiliser de la programmation dynamique, et plus précisément de la mémoïsation : on garde en mémoire les résultats des appels récursifs déjà effectués. Cela nécessitera un espace mémoire quadratique pour tout garder en mémoire.

On propose de garder les résultats en mémoire dans un tableau. On écrit une fonction auxiliaire qui ressemble à la précédente, mais qui prend en argument supplémentaire un tableau.

```
bool wythoff_memo(int m, int n, int** tab){
    if (tab[m][n] != 0) return tab[m][n] == 1;
    for (int k = 1; k <= m || k <= n; k++){
        if ((k <= m && !wythoff_memo(m - k, n, tab)) ||
            (k <= n && !wythoff_memo(m, n - k, tab)) ||
            (k <= m && k <= n && !wythoff_memo(m - k, n - k, tab))){
                tab[m][n] = 1;
                return true;
            }
    }
    tab[m][n] = 2;
    return false;
}
```

Le tableau contient 1 si on a déjà trouvé qu'il existe une stratégie gagnante pour le premier joueur, 2 si c'est pour le deuxième joueur et 0 si on n'a pas encore fait le calcul.

Dès lors, la fonction demandée crée le tableau, lance un appel à la fonction mémoïsée, puis libère la mémoire avant de renvoyer le résultat.

```
bool wythoff(int m, int n){
    int** tab = malloc((m + 1) * sizeof(int*));
    for (int i = 0; i < m + 1; i++){
        tab[i] = malloc((n + 1) * sizeof(int));
        for (int j = 0; j < m + 1; j++){
            tab[i][j] = 0;
        }
    }
    bool b = wythoff_memo(m, n, tab);
    for (int i = 0; i < m + 1; i++) free(tab[i]);
    free(tab);
    return b;
}
```

1.2 Nombre de Grundy

Question 7 On remarque que $\text{mex}(X) \leqslant |X|$. La valeur $|X|$ est atteinte si $X = \{0, 1, \ldots, |X| - 1\}$. Dans tous les autres cas, l'un des entiers compris entre 0 et $|X| - 1$ sera absent de X.

L'idée est alors de créer un tableau de booléens `tab` de taille $|X|$ et de parcourir X pour faire en sorte que `tab[x]` indique si x est présent dans X ou non. Une fois ce tableau calculé, on peut le parcourir à nouveau pour trouver le mex. On pense à vérifier que les éléments sont bien compris entre 0 et $|X|-1$ avant de modifier une case (il n'y a pas d'hypothèse sur X).

```
int mex(int* X, int n){
    bool* tab = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) tab[i] = false;
    for (int x = 0; x < n; x++){
        if (0 <= X[x] && X[x] < n) tab[X[x]] = true;
    }
    int mexX = n + 1;
    for (int x = 0; x < n; x++){
        if (!tab[x]) {
            mexX = x;
            break;
        }
    }
    return mexX;
}
```

Question 8 On remarque que si s est un puits, alors $g(s) = 0$ (car on cherche le mex d'un ensemble vide). Il s'agit d'un cas de base correct.

Question 9 On trouve les valeurs suivantes :

- $g(s) = 0$ pour $s \in \{(0,0),(1,1),(2,2)\}$;
- $g(s) = 1$ pour $s \in \{(0,1),(1,0),(3,2)\}$;
- $g(s) = 2$ pour $s \in \{(0,2),(2,0),(3,1)\}$;
- $g(s) = 3$ pour $s \in \{(1,2),(2,1),(3,0)\}$.

Question 10 Montrons ce résultat par récurrence sur k, la longueur maximale d'un chemin sortant de s_0 :

- si $k = 0$, alors s_0 est un puits et $g(s_0) = 0$, et 0 n'a effectivement pas de stratégie gagnante ;
- supposons le résultat établi jusqu'à $k \in \mathbb{N}$ fixé. Supposons que la longueur maximale d'un chemin sortant de s_0 soit $k+1$ et distinguons :
  - si $g(s_0) = 0$, alors chaque voisin t de s_0 vérifie $g(t) \neq 0$ (sinon $g(s_0)$ ne pourrait pas valoir 0). Par hypothèse de récurrence, il existe une stratégie gagnante pour le premier joueur depuis t. Ainsi, quel que soit le coup du premier joueur, le deuxième joueur a une stratégie gagnante, donc le premier joueur n'a pas de stratégie gagnante depuis s_0.
  - si $g(s_0) \neq 0$, alors il existe un voisin t de s_0 tel que $g(t) = 0$ (sinon on aurait $g(s_0) = 0$). Par hypothèse de récurrence, il n'existe pas de stratégie gagnante pour celui qui jouerait en premier depuis t. Ainsi, il existe une stratégie gagnante φ depuis s_0 pour le premier joueur, vérifiant $\varphi(s_0) = t$.

On conclut par récurrence.

Question 11 On commence par écrire une fonction qui calcule récursivement le nombre de Grundy, en mémoïsant les résultats dans le tableau `g`. Le principe est de calculer cette valeur pour les voisins, puis de calculer le mex.

```
int grundy_rec(graphe G, int* g, int s){
    if (g[s] != -1) return g[s];
    int* X = malloc(G.degres[s] * sizeof(int));
    for (int i = 0; i < G.degres[s]; i++){
        X[i] = grundy_rec(G, g, G.adj[s][i]);
    }
    g[s] = mex(X, G.degres[s]);
    free(X);
    return g[s];
}
```

Dès lors, on obtient la fonction `grundy` suivante en calculant les valeurs de Grundy pour chaque sommet :

```c
int* grundy(graphe G){
    int* g = malloc(G.n * sizeof(int));
    for (int s = 0; s < G.n; s++) g[s] = -1;
    for (int s = 0; s < G.n; s++) grundy_rec(G, g, s);
    return g;
}
```

Question 12 La fonction `mex` est de complexité linéaire. Il y aura au plus un appel récursif à `grundy_rec` par sommet. Cet appel ne fait qu'un nombre d'opérations proportionnel au degré de s. Ainsi, la complexité totale est en $\mathcal{O}(|S| + |A|)$ pour un graphe $G = (S, A)$.

2 Automates à repli

Question 13 On suit naïvement la définition. On pense à faire un décalage de 1 dans les indices, pour distinguer la numérotation à partir de 0 en C et à partir de 1 dans l'énoncé.

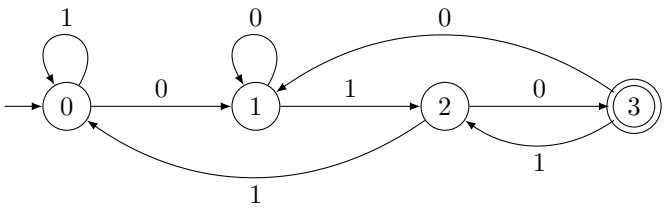
```c
void recherche_naive(char* u, char* v){
    int k = strlen(u);
    int n = strlen(v);
    for (int i = 1; i <= n; i++){
        int j = 0;
        while (j < k && u[k - j - 1] == v[i - j - 1]){
            j++;
        }
        if (j == k) printf("%d\n", i);
    }
}
```

Question 14 Le calcul des longueurs se fait en $\mathcal{O}(k+n)$. La double boucle se fait dans le pire cas en $\mathcal{O}(nk)$ (par exemple pour $u, v \in a^*$).

2.1 Automates finis déterministes à repli

Question 15 On remarque que par définition, la suite $(\rho^j(q))_{j\in\mathbb{N}}$ est strictement décroissante tant que $\rho^j(q) \neq 0$. L'ensemble $\mathbb{N}$ étant bien fondé, on en déduit que la suite est ultimement constante égale à 0. En particulier, $\rho^k(q) = 0$ pour tout état $q \in Q_{\mathcal{A}}$. Comme pour tout $a \in \Sigma$, $\delta(0, a)$ est bien défini, on en déduit que $\delta(\rho^k(q), a)$ est toujours défini, ce qui montre l'existence de l'entier i demandé.

Question 16 L'automate suivant convient.



L'idée est la suivante : on écrit les transitions déjà existantes, et on rajoute les transitions manquantes en se basant sur le repli :

– pour la lettre 0 depuis l'état 1, on se replie à l'état 0, puis on lit un 0 pour revenir en 1 ;
– pour la lettre 1 depuis l'état 2, on se replie à l'état 0, puis on lit un 1 pour rester en 0 ;
– pour la lettre 0 depuis l'état 3, on se replie à l'état 1, puis à l'état 0, puis on revient en 1 en lisant 0 ;
– pour la lettre 1 depuis l'état 3, on se replie à l'état 1, puis on arrive en 2.

Question 17 Le langage est celui des mots qui finissent par 010, c'est-à-dire $L(\mathcal{A}_1) = \Sigma^*\{010\}$.

Question 18 On commence par copier complètement l'automate grâce à la fonction précédente. Ensuite, dans l'ordre croissant des états q, pour chaque lettre a pour laquelle une transition n'est pas définie, on copie la transition $\delta(\rho(q), a)$ (qui a éventuellement été modifié dans un précédent passage dans la boucle). La complexité de la fonction `copie_afdr` est a priori en $\mathcal{O}(k)$ ($k+1$ copies de tableaux de taille 256, plus deux copies de tableaux de taille $k+1$). De par la taille des boucle `for` dans la fonction ci-dessous, et parce que les opérations qui sont à l'intérieur sont en temps constant, on a bien à nouveau une complexité $\mathcal{O}(k)$.

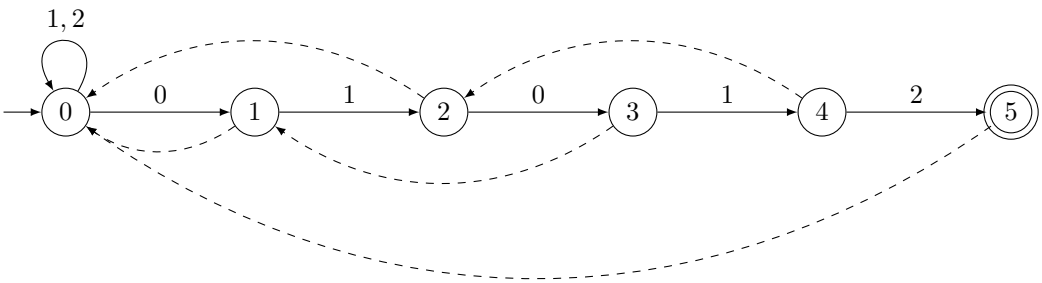
```
afdr enleve_repli(afdr A){
    afdr AFDC = copie_afdr(A);
    for (int q = 1; q < A.k; q++){
        for (int a = 0; a < 256; a++){
            if (A.delta[q][a] == -1){
                AFDC.delta[q][a] = AFDC.delta[A.rho[q]][a];
            }
        }
    }
    return AFDC;
}
```

Question 19 On se contente de lire les lettres du mot u une à une dans l'AFDC $\mathcal{A}$, et de vérifier pour chaque lettre si l'état dans lequel on se trouve est final ou non. On pense bien à tester le mot vide.

```
void occurrences(afdr A, char* u){
    int q = 0;
    if (A.F[q]) printf("%d\n", 0);
    for (int i = 0; u[i] != '\0'; i++){
        q = A.delta[q][u[i]];
        if (A.F[q]) printf("%d\n", i + 1);
    }
}
```

2.2 Automate de Knuth-Morris-Pratt

Question 20 On obtient l'automate suivant :



Question 21 $\mathcal{A}_u^{\text{KMP}}$ reconnait le langage $\Sigma^*\{u\}$.

Question 22 On utilise la formule précédente pour remplir la fonction de repli. Dans la fonction ci-dessus, on commence par initialiser toutes les transitions depuis 0 vers 0 et les autres étant des blocages.

On boucle ensuite sur un indice i (correspondant à l'indice de la lettre en cours de lecture). On traite à part la première lettre du mot (car la formule précédente n'est pas vérifiée), en posant $\delta(0, u_1) = 1$ et $\rho(1) = 0$, et si $i \geqslant 2$, on pose $\delta(i-1, u_i) = i$ et on cherche la valeur de j_i (à l'aide d'une boucle `while`) pour remplir la valeur de $\rho(i)$.

```
afdr KMP(char* u){
    afdr A;
    A.k = 1 + strlen(u);
    A.F = malloc(A.k * sizeof(bool));
    A.delta = malloc(A.k * sizeof(int*));
    A.rho = malloc(A.k * sizeof(int));
    for (int q = 0; q < A.k; q++){
        A.F[q] = q == A.k - 1;
        A.delta[q] = malloc(256 * sizeof(int));
        for (int a = 0; a < 256; a++){
            A.delta[q][a] = -1;
            A.delta[0][a] = 0;
        }
        A.rho[q] = 0;
    }
    A.delta[0][u[0]] = 1;
    for (int i = 1; i < A.k - 1; i++){
        A.delta[i][u[i]] = i + 1;
        int q = A.rho[i];
        while (A.delta[q][u[i]] == -1) q = A.rho[q];
        A.rho[i + 1] = A.delta[q][u[i]];
    }
    return A;
}
```

Question 23 On sait que $\rho(i) = \delta(\rho^{j_i}(\rho(i-1)), u_i)$. Or, $\delta(\rho^{j_i}(\rho(i-1)), u_i)$ vaut 0 ou $\rho^{j_i}(\rho(i-1)) + 1$, donc $\rho(i) \leqslant \rho^{j_i}(\rho(i-1)) + 1$. Enfin, comme $\rho(x) < x$ si $x \neq 0$, par une récurrence rapide, on montre que $\rho^\ell(x) \leqslant x - \ell$ tant que $\rho^{\ell-1}(x) \neq 0$.

Finalement, on a $\rho(i) \leqslant \rho^{j_i}(\rho(i-1)) + 1 \leqslant \rho(i-1) + 1 - j_i$.

On en déduit que $j_i \leqslant \rho(i-1) - \rho(i) + 1 \leqslant 1$ (car $\rho(i) \geqslant \rho(i-1)$, soit finalement $\sum_{i=1}^{k} j_i \leqslant k = \mathcal{O}(k)$.

Question 24 La création des tableaux se fait en $\mathcal{O}(k)$. La dernière boucle `for` fait k itérations et pour chaque itération, on a une boucle de taille j_i et des opérations en temps constant. On en déduit que le passage pour i s'exécute en temps $\mathcal{O}(\sum_{i=1}^{k} j_i) = \mathcal{O}(k)$. La complexité totale est donc en $\mathcal{O}(k)$.

Question 25 On construit l'automate KMP associé à u et on utilise la fonction `occurrences` écrite précédemment en ayant au préalable complété l'automate.

```
void recherche_KMP(char* u, char* v){
    afdr A = KMP(u);
    afdr AFDC = enleve_repli(A);
    occurrences(AFDC, v);
    liberer_afdr(A);
    liberer_afdr(AFDC);
}
```

On a une complexité en $\mathcal{O}(k+n)$ au lieu de $\mathcal{O}(k \times n)$ pour la recherche naïve.

⋆ ⋆ ⋆