Devoir maison n°1

Corrigé

⋆ ⋆ ⋆

1 Vote par préférence

1.1 Premier exemple

Question 1 On commence par écrire une fonction qui détermine qui gagne un duel pour un seul bulletin :

```
let rec duel_bul i j = function
    | []     -> failwith "Bulletin incorrect"
    | x :: q -> if x = i then 1
                else if x = j then -1
                else duel_bul i j q;;
```

On peut alors l'appliquer à chaque bulletin et sommer le tout :

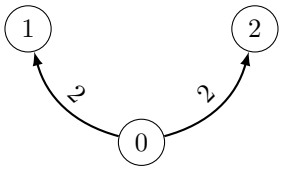
```
let duel i j u =
    let score = ref 0 in
    List.iter (fun bul -> score := !score + duel_bul i j bul) u;
    !score;;
```

On aurait pu utiliser `List.fold_left` pour faire ce calcul :

```
let duel i j u =
    List.fold_left (fun score bul -> score + duel_bul i j bul) 0 u;;
```

1.2 Deuxième exemple

Question 2 On obtient le graphe et la matrice suivants :



$$M = \begin{pmatrix} 0 & 2 & 2 \\ -2 & 0 & 0 \\ -2 & 0 & 0 \end{pmatrix}$$

1.3 Construction du graphe de préférence

Question 3 Par convention, les coefficients diagonaux sont nuls. De plus, pour $i \neq j$, on remarque que `duel i j u` $= -$(`duel j i u`). En effet, si i emporte a victoires sur j, et j emporte b victoire sur i, alors `duel i j u` $= a - b$ et `duel j i u` $= b - a$. La matrice est bien antisymétrique. De plus, on remarque que la parité de chaque coefficient est égale à p, le nombre de votants. En effet, $p = a + b$ qui est de la même parité que $a - b$.

Question 4 On se contente de créer la matrice et de faire un appel à `duel` pour chaque coefficient non diagonal.

```
let depouillement n u =
    let mat = Array.make_matrix n n 0 in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            if i <> j then mat.(i).(j) <- duel i j u
        done
    done;
    mat;;
```

À noter, on aurait pu profiter de l'antisymétrie pour faire deux fois moins d'appels à `duel` (mais cela ne change pas la complexité asymptotiquement).

1.4 Théorème de McGarvey

Question 5 On veut que i gagne deux fois contre j, mais que tous les autres duels soient des ex æquo. Notons $(c_0, c_1, \ldots, c_{n-3})$ les valeurs de $[\![0, n-2]\!] \setminus \{i, j\}$, dans un ordre arbitraire. On pose :

$$U_{i,j,n} = \{(i, j, c_0, c_1, \ldots, c_{n-3}), (c_{n-3}, c_{n-4}, \ldots, c_0, i, j)\}$$

On a bien le résultat attendu.

Question 6 On a $M_3 = M_1 + M_2$. En effet, les scores sont additifs pour chaque nouveau bulletin ajouté.

Question 7 Notons $kU = \underbrace{U \cup U \cup \ldots \cup U}_{k \text{ fois}}$ pour U une urne quelconque et $k \in \mathbb{N}^*$. Soit M une matrice antisymétrique à coefficients pairs. On pose $U = \bigcup_{m_{ij}>0} \frac{m_{ij}}{2} E_{i,j,n}$. Alors, par récurrence en utilisant les deux questions précédentes, on obtient $\mathrm{Mat}(U) = M$.

Question 8 On commence par écrire une fonction qui crée une urne élémentaire $U_{i,j,n}$:

```
let urne_elem i j n =
    let lst = List.init n (fun x -> x) in
    let u = List.filter (fun x -> x <> i && x <> j) lst in
    [i :: j :: u; List.rev u @ [i; j]];;
```

Ici, on a d'abord construit la liste de tous les candidats, puis on y a enlevé i et j, puis on construit l'urne à deux bulletins. Avec cette fonction, on peut construire l'urne finale :

```
let mcgarvey m =
    let n = Array.length m in
    let u = ref [] in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            if m.(i).(j) > 0 then
               let uijn = urne_elem i j n in
               for k = 0 to m.(i).(j) / 2 - 1 do
                   u := uijn @ !u
               done
        done
    done;
    !u;;
```

Pour chaque coefficient strictement positif, on crée (une seule fois) l'urne élémentaire correspondant aux indices, et on l'ajoute à l'urne `u` autant de fois que nécessaire. À noter, la complexité augmente si on écrit `!u @ uijn` au lieu de `uijn @ !u` (car `uijn` est toujours de taille 2).

Question 9 La fonction `urne_elem` est en complexité $\mathcal{O}(n)$ (chacun des appels à `List.init`, `List.filter`,

`List.rev` et le `@`). Dans la fonction `mcgarvey`, à l'intérieur des deux boucles externes, on fait un appel à `urne_elem`, suivi d'une boucle de taille $\mathcal{O}(q)$ qui fait une opération en temps constant. On a donc une complexité totale en $\mathcal{O}(n^2(n+q))$.

2 Recherche du vainqueur

2.1 Manipulation de matrices

Question 10 On commence par allouer la matrice, puis pour chaque ligne, on l'alloue et on l'initialise.

```
int** copie_matrice(int** M, int n){
    int** M2 = malloc(n * sizeof(int*));
    for (int i=0; i<n; i++){
        M2[i] = malloc(n * sizeof(int));
        for (int j=0; j<n; j++){
            M2[i][j] = M[i][j];
        }
    }
    return M2;
}
```

Question 11 On pense à libérer chaque ligne avant de libérer la matrice.

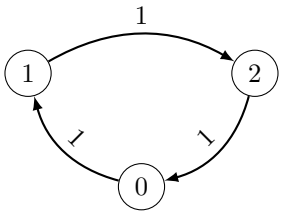
```
void liberer_matrice(int** M, int n){
    for (int i=0; i<n; i++){
        free(M[i]);
    }
    free(M);
}
```

2.2 Vainqueur de Condorcet

Question 12 On commence par créer le tableau de booléens (pas besoin de l'initialiser, on va modifier chaque case). Pour chaque ligne, on garde en mémoire un entier j correspondant à l'indice maximal dont tous les éléments précédents sont positifs ou nul. Si cet indice atteint la taille de la matrice, le candidat de la ligne correspondante est un gagnant de Condorcet.

```
bool* condorcet(int** M, int n){
    bool* tab = malloc(n * sizeof(bool));
    for (int i=0; i<n; i++){
        int j = 0;
        while (j<n && M[i][j] >= 0){
            j++;
        }
        tab[i] = (j == n);
    }
    return tab;
}
```

Question 13 On pose $\{(0,1,2),(1,2,0),(2,0,1)\}$. On obtient le graphe de préférence et la matrice suivants :



$$M = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix}$$

2.3 Graphe intermédiaire de Schultze

Question 14 Soit c un chemin de poids maximal dans le graphe de préférence, dont le multi-ensemble des arêtes est A_c. Notons c^* le chemin c auquel on a retiré tous les cycles, et A^* son ensemble d'arêtes. Comme $A^* \subseteq A_c$, on a $\min\{P(a) \mid a \in A^*\} \geqslant \min\{P(a) \mid a \in A_c\}$ (où P désigne la fonction de poids), donc $P(c^*) \geqslant P(c)$. On en déduit que c^* est également de poids maximal, ce qui conclut.

Question 15 En concaténant un chemin de poids maximal de i à j et un chemin de poids maximal de j à k, on obtient un chemin de i à k. Or le poids d'un tel chemin est $\min(I[i,j], I[j,k])$, et comme c'est un chemin de i à k, cette valeur est $\leqslant I[i,k]$.

Question 16 Comme pour l'algorithme de Floyd-Warshall, on construit une suite de matrices $I^(k)$ telles que :

- $I^(0) = M$;
- pour $k \in [\![0, n-1]\!]$ et $i, j \in [\![0, n-1]\!]^2$,

$$I^{(k+1)}[i,j] = \begin{cases} I^{(k)}[i,j] & \text{si } k = i \text{ ou } k = j \\ \max(I^{(k)}[i,j], \min(I^{(k)}[i,k], I^{(k)}[k,j]) & \text{sinon} \end{cases}$$

On remarque alors que $I^{(k)}[i,j]$ correspond au poids maximal d'un chemin de i à j, dont les sommets intermédiaires sont tous strictement inférieurs à k. En effet, c'est vrai pour $k = 0$, car le seul chemin de i à j qui ne passe par aucun sommet intermédiaire est l'arête (i,j), et si c'est vrai pour $k \in [\![0, n-1]\!]$, alors c'est vrai pour $k+1$ d'après la question précédente. La matrice I cherchée est donc $I^{(n)}$.

On commence par créer deux fonctions utilitaires de `min` et `max`.

```
int min(int a, int b){
    return a<b?a:b;
}

int max(int a, int b){
    return a>b?a:b;
}
```

En adaptant l'algorithme de Floyd-Warshall, on obtient :

```
int** intermediaire(int** M, int n){
    int** M1 = copie_matrice(M, n);
    for (int k=0; k<n; k++){
        int** M2 = copie_matrice(M1, n);
        for (int i=0; i<n; i++){
            for (int j=0; j<n; j++){
                if (i != k && j != k){
                    M1[i][j] = max(M2[i][j], min(M2[i][k], M2[k][j]));
                }
            }
        }
        liberer_matrice(M2, n);
    }
    return M1;
}
```

L'idée est que pour chaque valeur de k, on crée une copie `M2` de la matrice `M1`, qu'on utilise pour modifier `M1`, en accord avec la formule précédente. On pense bien à libérer la mémoire inutile.

Question 17 La complexité spatiale est en $\mathcal{O}(n^2)$ (les deux matrices). La complexité temporelle est en $\mathcal{O}(n^3)$, correspondant aux trois boucles imbriquées dans lesquelles on fait des opérations en temps constant (la création de `M2` est en $\mathcal{O}(n^2)$, identique aux deux boucles qui suivent).

Question 18 L'algorithme de Floyd-Warshall repose sur le fait qu'on peut déduire le poids (ici, le minimum) de la concaténation de deux chemins en connaissant le poids de chacun. C'est ce qui permet de l'adapter ici.

L'algorithme de Dijkstra repose sur le fait que s'il existe un chemin de i à k de poids p, alors un chemin de i à j de poids $p' > p$ ne peut pas être le début d'un chemin de poids $\leqslant p$ de i à k. Or, quand le poids d'un chemin est le minimum des arêtes, cette propriété devient fausse. On ne peut donc pas l'adapter ici (ou alors avec beaucoup de modifications).

2.4 Graphe de préférence de Schulze

Question 19 On a $S[i,j] = I[i,j] - I[j,i] = -(I[j,i] - I[i,j] = -S[j,i]$, donc la matrice est bien antisymétrique. De plus, tous les $I[i,j]$ sont de même parité, car ils correspondent aux poids d'une arête du graphe de préférence, dont la matrice M a tous ses coefficients de même parité.

Question 20 On se contente d'appliquer la définition.

```
int** graphe_schulze(int** I, int n){
    int** S = copie_matrice(I, n);
    for (int i=0; i<n; i++){
        for (int j=0; j<n; j++){
            S[i][j] -= I[j][i];
        }
    }
    return S;
}
```

2.5 Vainqueur de Schulze

Question 21 À nouveau, on applique la définition. On pense à libérer la mémoire inutile.

```
bool* schulze(int** M, int n){
    int** I = intermediaire(M, n);
    int** S = graphe_schulze(I, n);
    bool* vainqueur = condorcet(S, n);
    liberer_matrice(I, 3);
    liberer_matrice(S, 3);
    return vainqueur;
}
```

Question 22 La fonction `graphe_schulze` est en $\mathcal{O}(n^2)$ (création de la matrice et deux boucles de taille n imbriquées)k. La fonction `condorcet` est également en $\mathcal{O}(n^2)$ (parcours de tous les coefficients de la matrice). La complexité totale est celle de `intermediaire`, soit $\mathcal{O}(n^3)$.

Question 23 Supposons $iR_S j$ et $jR_S k$. On a donc $I[i,j] > I[j,i]$ et $I[j,k] > I[k,j]$. Distinguons :

- si $I[i,j] > I[j,k]$, on a, en utilisant Q15 :
  - $\min(I[i,j], I[j,k]) = I[j,k] \leqslant I[i,k]$;
  - $\min(I[k,i], I[i,j]) \leqslant I[k,j] < I[j,k] < I[i,j]$, soit $I[k,i] < I[i,j]$;

* finalement, $I[k,i] = \min(I[k,i], I[i,j]) \leqslant I[k,j] < I[j,k] < I[i,k]$.

On a bien iR_Sk ;

– si $I[i,j] \leqslant I[j,k]$, on a :

* $\min(I[i,j], I[j,k]) = I[i,j] \leqslant I[i,k]$;
* $\min(I[j,k], I[k,i]) \leqslant I[j,i] < I[i,j] \leqslant I[j,k]$, soit $I[k,i] < I[j,k]$;
* finalement, $I[k,i] = \min(I[j,k], I[k,i]) \leqslant I[j,i] < I[i,j] = \min(I[i,j], I[j,k]) \leqslant I[i,k]$.

À nouveau, iR_Sk.

Question 24 Supposons par l'absurde qu'il n'existe pas de vainqueur de Schulze et soit i_0 un candidat. Comme i_0 n'est pas vainqueur de Condorcet dans le graphe des préférences de Schulze, il existe $i_1 \neq i_0$ tel que $i_1R_Si_0$. De la même manière, on définit i_2, i_3, ... Par le principe des tiroirs, il existe $k < k'$ tels que $i_k = i_{k'}$. Mais alors, par la question précédente, on obtient $i_{k'}R_Si_k$, ce qui est absurde.

⋆ ⋆ ⋆