Tournois et pronostics (X psi-pt 2014)

Partie I. Tournois

Question 1. La condition (3) s'interprète comme « *les $n-1$ derniers éléments de la représentation d'un tournoi sont les matchs réels opposant deux adversaires différents* ».
La condition (4) s'interprète comme « *les matchs réels ne peuvent opposer que des joueurs débutant le tournoi ou vainqueur d'un match précédent* ». (Les joueurs du match ne peuvent être les vainqueurs d'un match n'ayant pas encore eu lieu.)
La condition (5) s'interprète comme « *tout joueur participe au tournoi ; tout vainqueur (à l'exception de celui du dernier match) participe à un autre match* » ; (Tout joueur ayant perdu un match est définitivement éliminé.)

Question 2. La fonction qui suit calcule $s = \text{card}\left\{i \in [\![k+1, 2n-2]\!] \;\middle|\; k \in \{T[i][0], T[i][1]\}\right\}$ puis détermine si $s = 1$.

```
def EstVraiCondition5(T, k):
    s = 0
    for i in range(k+1, 2*n-1):
        if k in T[i]:
            s += 1
    return s == 1
```

Puisque $T[i]$ est de longueur constante (égale à 2), le test « `k in T[i]` » est de coût constant, et la complexité de cette fonction est en $O(2n-k)$.

Question 3. Chacune des cinq conditions est vérifiée à la suite l'une de l'autre :

```
def EstUnTournoi(T):
    for k in range(2*n-1):
        if T[k][1] < T[k][0]:
            return False
    for k in range(n):
        if T[k][0] != k or T[k][1] != k:
            return False
    for k in range(n, 2*n-1):
        if T[k][0] == T[k][1]:
            return False
    for k in range(n, 2*n-1):
        if not (0 <= T[k][0] < k and 0 <= T[k][1] < k):
            return False
    for k in range(2*n-2):
        if not EstVraiCondition5(T, k):
            return False
    return True
```

Notez au passage (cf condition (4)) que si `c1` et `c2` sont deux comparaisons, la syntaxe `x c1 y c2 z` est équivalente à `x c1 y and y c2 z`.
La coût de la vérification des quatre premières comparaisons est à l'évidence un $O(n)$. Le coût de la vérification de la cinquième comparaison est un $O\left(\sum_{k=0}^{2n-3}(2n-k)\right) = O(2n^2+n-3) = O(n^2)$.
La complexité de la fonction `EstUnTournoi` est donc en $O(n^2)$.

Question 4.

```
def EstUnOracle(O):
    for i in range(n):
        if not O[i][i]:
            return False
        for j in range(i+1, n):
            if O[i][j] == O[j][i]:
                return False
    return True
```

On effectue au plus $\frac{n(n-1)}{2}$ comparaisons donc la complexité de cette fonction est en $O(n^2)$.

Question 5. La fonction qui suit utilise un tableau M destiné à contenir le résultat de chaque match. On commence par remplir ce tableau avec les n matchs triviaux, puis le reste du tableau à l'aide des résultats des matchs précédents et de l'oracle.

```
def Vainqueur(T, O):
    M = [None] * (2*n-1)
    for k in range(n):
        M[k] = k
    for k in range(n, 2*n-1):
        i, j = M[T[k][0]], M[T[k][1]]
        if O[i][j]:
            M[k] = i
        else:
            M[k] = j
    return M[-1]
```

Chacune des opérations effectuées dans les deux boucles est de coût constant, donc la complexité de cette fonction est en $O(n)$ (assorti d'un coût spatial — le tableau M — lui aussi en $O(n)$).

Partie II. Vainqueurs potentiels

Question 6. Pour donner un exemple d'oracle n'ayant qu'un seul vainqueur potentiel, il suffit que parmi les participants l'un d'eux soit assuré de battre tous les autres, autrement dit que la matrice O contienne une ligne ne comportant que la valeur `True`.
Si maintenant on considère un oracle dans lequel le premier joueur gagne tous ses matchs sauf celui contre le deuxième joueur et dans lequel le deuxième joueur perd tous ses matchs à l'exception de celui contre le premier joueur, alors ces deux joueurs sont tous deux des vainqueurs potentiels. En effet, pour que le premier joueur gagne le tournoi il suffit que le second joueur ne le rencontre pas lors de son premier match ; pour que le second joueur gagne il suffit qu'il n'entre dans le tournoi que pour participer à la finale.

Question 7. La fonction qui suit retourne le résultat du tournoi défini par : $\forall k \in [\![n, 2n-2]\!]$, $T[k] = [k-n, k-1]$ (le joueur $n-1$ rencontre le joueur 0, le vainqueur rencontre le joueur 1, le vainqueur rencontre le joueur 2, etc.). On obtient ainsi un vainqueur potentiel.

```
def UnVainqueur(O):
    T = [None] * (2*n-1)
    for k in range(n):
        T[k] = [k, k]
    for k in range(n, 2*n-1):
        T[k] = [k-n, k-1]
    return Vainqueur(T, O)
```

La création du tournoi T et l'appel à la fonction `Vainqueur` ont tous deux un coût en $O(n)$ donc la complexité de cette fonction est aussi en $O(n)$.

Question 8. Considérons un vainqueur potentiel i et un joueur j qui gagne lors d'un affrontement avec i. Il existe donc un tournoi T remporté par i et dans lequel i n'affronte pas j. Le joueur j a donc perdu contre un certain joueur $k \neq i$. Supprimons la partie T_1 du tournoi ayant vu la victoire de j juste avant sa défaite contre k, et déplaçons cette partie après la finale de T pour faire rencontrer i et j en finale. On obtient un nouveau tournoi dans lequel j est vainqueur, donc j est aussi un vainqueur potentiel pour l'oracle O.

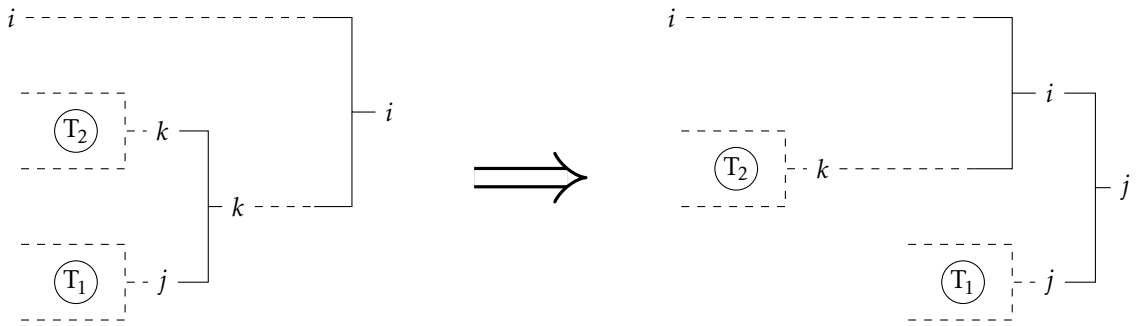

Figure 1 – La modification à apporter à T pour que j soit vainqueur.

Question 9. Considérons un tournoi T et k le dernier match gagné par un joueur de X, joueur que nous notons x.
Si $k < 2n-2$, la propriété (5) affirme l'existence d'un match ultérieur $k' > k$ auquel participe x. Par définition de k, x a perdu ce match. Mais les joueurs qui n'appartiennent pas à X ne peuvent gagner contre x donc son vainqueur y appartient lui aussi à X, ce qui contredit la définition de k.
De ceci il résulte que $k = 2n-2$, autrement dit que x a gagné le tournoi.
Nous venons de montrer que tous les tournois sont gagnés par un élément de X, qui contient donc tous les vainqueurs potentiels.

Question 10. Nous allons faire évoluer un ensemble X de vainqueurs potentiels, d'abord en y plaçant un élément à l'aide de la question 7, puis en y ajoutant tous les joueurs qui gagnent contre un élément quelconque de X. Une fois ce processus achevé, tous les éléments de X sont des vainqueurs potentiels (question 8), et tous les joueurs qui n'y sont pas rentrés n'en sont pas (question 9). X est donc bien alors l'ensemble des vainqueurs potentiels.

```
def VainqueursPotentiels(O):
    v = [False] * n
    x = UnVainqueur(O)
    v[x] = True
    L = [x]
    while not len(L) == 0:
        y = L.pop()
        for i in range(n):
            if not v[i] and O[i][y]:
                v[i] = True
                L.append(i)
    return v
```

La liste L contient les vainqueurs potentiels pour lesquels on a pas encore ajouté dans X les joueurs qui les battent.
Le coût de cette fonction est un $O(pn)$ où p désigne le nombre de vainqueurs potentiels de O (pour chacun d'eux, on effectue une recherche linéaire pour déterminer les adversaires qui les battent) ; dans le pire des cas on peut donc affirmer que le coût est un $O(n^2)$.

Partie III. Pronostics

Question 11. Il s'agit de parcourir l'arborescence située avant $T[k][0]$. On utilise pour cela une liste M contenant initialement $T[k][0]$, et tant que cette liste n'est pas vide, on en sort un élément x. Si x désigne un joueur, ce joueur peut être le premier joueur du match k ; si x désigne un match, on ajoute $T[x][0]$ et $T[x][1]$ à la liste M.

```
def PremierJoueurPossible(T, k):
    L = [False] * n
    M = [T[k][0]]
    while len(M) > 0:
        x = M.pop()
        if x < n:
            L[x] = True
        else:
            M.append(T[x][0])
            M.append(T[x][1])
    return L
```

Question 12. Il parait difficile de modifier en place le vecteur V, c'est pourquoi on utilise une copie V′ de V qui calcule les nouvelles probabilités. Une fois toutes ces valeurs calculées on met à jour le vecteur V.

```
def MiseAJour(T, P, k, V):
    Vprime = V.copy()
    PJP = PremierJoueurPossible(T, k)
    SJP = SecondJoueurPossible(T, k)
    for i in range(n):
        if PJP[i]:
            s = 0
            for j in range(n):
                if SJP[j]:
                    s += V[j] * P[i][j]
            Vprime[i] *= s
        elif SJP[i]:
            s = 0
            for j in range(n):
                if PJP[j]:
                    s += V[j] * P[i][j]
            Vprime[i] *= s
    for i in range(n):
        V[i] = Vprime[i]
```

Question 13. Il reste à appliquer cette fonction à chaque étape du tournoi :

```
def ChancesDeVictoire(T, P):
    V = [1] * n
    for k in range(n, 2*n-1):
        MiseAJour(T, P, k, V)
    return V
```