Proposition de corrigé du TP8 : Théorie des jeux, apprentissage

1 Jeu du morpion

1.1 Premiers outils

1. On utilise deux boucles pour obtenir l'affichage demandé :

```
def plateau(jeu):
    print('---')
    for l in range(3):
        ligne = ''
        for c in range(3):
            ind = 3*l + c
            case = jeu[ind]
            ligne += case
        print(ligne)
```

2. On compte le nombre de cases vides avec une boucle et retourne le booléen demandé :

```
def est_plein(jeu):
    t = 0
    for e in jeu:
        if e!='␣':
            t += 1
    return t == 9
```

3. On donne une structure conditionnelle (on ne renvoie rien lorsque i n'est pas 1 ou 2) :

```
def nbr2str(i):
    if i==1:
        return "X"
    elif i==2:
        return "O"
    else:
        ERREUR
```

4. La fonction est très similaire à la question précédente :

```
def str2nbr(ch):
    if ch=='X':
        return 1
    elif ch=='O':
        return 2
    else:
        ERREUR
```

5. Le nombre de conditions est important. La fonction suivante traite effectivement tous les cas :

```
def est_gagnant(jeu):
    c11,c12,c13,c21,c22,c23,c31,c32,c33 = jeu
    if c11 == c21 == c31 != '␣':
        return True,str2nbr(c11)
    elif c12 == c22 == c32 != '␣':
        return True,str2nbr(c12)
    elif c13 == c23 == c33 != '␣':
        return True,str2nbr(c13)
    elif c11 == c12 == c13 != '␣':
        return True,str2nbr(c11)
    elif c21 == c22 == c23 != '␣':
        return True,str2nbr(c21)
    elif c31 == c32 == c33 != '␣':
        return True,str2nbr(c31)
    elif c11 == c22 == c33 != '␣':
        return True,str2nbr(c11)
    elif c31 == c22 == c13 != '␣':
        return True,str2nbr(c31)
    else:
        return False,None
```

6. On vérifie si l'un des cas suivants est réalisé : le jeu est plein ou un joueur est gagnant.

```
def est_fini(jeu):
    Cond_plein = est_plein(jeu)
    Cond_gain,j = est_gagnant(jeu)
    return Cond_plein or Cond_gain
```

1.2 Graphe du jeu

7. On parcourt tous les vides possibles de la chaîne de caractères et on remplit en conséquence avec le bon joueur :

```
def coups(pos):
    jeu,j = pos
    adv = 3-j
    Res = []
    if not est_fini(jeu):
        for i in range(9):
            e = jeu[i]
            if e == '␣':
                deb = jeu[:i]
                fin = jeu[i+1:]
                Sol = deb + nbr2str(j) + fin
                Res += [(Sol,adv)]
    return Res
```

8. On part d'une position et on utilise la fonction précédente pour remplir peu à peu le graphe :

```
def graphe(x0):
    G = {}
    G[x0] = coups(x0)
    file=[]
    file.append(x0)
    while len(file) != 0:
        x = file.pop()
        Lc = G[x] # Déjà créé, gagne du temps vs coups(c)
        for c in Lc:
            if c not in G:
                file.append(c)
                G[c] = coups(c)
    return G
```

1.3 Attracteurs

9. On cherche ici les états gagnants de façon naïve. On parcourt le graphe et on utilise `est_gagnant`.

```
def init_attracteur(G):
    d1 =[]
    for x in G:
        jeu,joueur = x
               if est_gagnant(jeu)==(True,1):
                      d1.append(x)
    return d1
```

10. On commence par recopier les fonctions du cours selon la définition par récurrence de l'attracteur.

```
def Existence(G,d1,x):
    '''Renvoie True si au moins un successeur de x est True dans d1'''
    L_succ = G[x]
    for succ in L_succ:
        if succ in d1:
            return True
    return False
```

```
def Pourtout(G,d1,x):
    '''Renvoie True si tous les successeurs de x sont True dans d1'''
    L_succ = G[x]
    if len(L_succ) == 0:
        return False
    else:
        Res = True
        for succ in L_succ:
            Res = Res and (succ in d1)
        return Res
```

On donne finalement l'attracteur comme dans le cours selon la nomenclature prévue ici.

```
def attracteur(G):
    d1=init_attracteur(G)
    Changement = True
    while Changement:
        Changement = False
        for x in G:
            jeu,joueur = x
            if x not in d1: # Rq
                if joueur == 1 and Existence(G,d1,x):
                    d1.append(x)
                    Changement=True
                elif joueur==2 and Pourtout(G,d1,x):
                    d1.append(x)
                    Changement =True
    return d1
```

2 Classification et continents

2.1 Représentation graphique des villes

1. On réutilise ce qui a été fait dans le TP sur les bases de données (avec le chemin adapté) (et on donne les différents modules pour la suite) :

```
import sqlite3
import os
import matplotlib.pyplot as plt
import numpy as np
```



```
from random import sample

#ici le chemin d'acces est a modifier
os.chdir('chemin')
baseDeDonnees = sqlite3.connect('mondial.db')
curseur = baseDeDonnees.cursor()
def requete(char):
    curseur.execute(char)
    return list(curseur)

liste_villes = requete("SELECT␣*␣FROM␣city")
```

2. On récupère simplement une partie de la requête précédente :

```
longitude = []
latitude = []
for ville in liste_villes:
    latitude.append(ville[4])
    longitude.append(ville[5])
```

3. On utilise simplement la commande :

```
plt.plot(longitude, latitude,"o")
plt.show()
```

2.2 Clustering

4. Compte-tenu de la forme des listes on compare tous les éléments des listes qui comptent deux coordonnées :

```
def listes_egales(l1,l2):
    if not len(l1)==len(l2):
        return False
    for i in range(len(l1)):
        for j in range(len(l1[i])):
            if l1[i][j] != l2[i][j]:
                return False
    return True
```

5. On utilise la méthode du candidat pour déterminer cet indice :

```
def indice_plus_proche(X, liste):
    m=np.inf
    for i in range(len(liste)):
        Y=liste[i]
        M=np.sqrt(np.sum((X-Y)**2))
        if M<m:
            m=M
            ind_plus_proche=i
    return ind_plus_proche
```

6. On s'appuie sur le cours en utilisant la suggestion sur la fonction `sample` :

```
def mise_a_jour_centres(centres,liste):
    k=len(centres)
    nouveaux_centres=[np.zeros(2) for i in range(k)]
    cardinal_classes=[0 for _ in range(k)]
    for X in liste:
        nouvelle_classe_de_X=indice_plus_proche(X,centres)
        nouveaux_centres[nouvelle_classe_de_X]+=X
        cardinal_classes[nouvelle_classe_de_X]+=1
    nouveaux_centres=[nouveaux_centres[i]/cardinal_classes[i] for i in range(k)]
    return nouveaux_centres
```

```
def kmoyennes(liste,k):
    centres = sample(liste,k)
    nouveaux_centres=mise_a_jour_centres(centres,liste)
    while not (listes_egales(centres, nouveaux_centres)):
        centres=nouveaux_centres
        nouveaux_centres=mise_a_jour_centres(centres,liste)
    return centres
```

7. On utilise le code :

```
def dessine_donnees_avec_moyennnes(listeX, centres):
    formes=['k', 'g', 'b', 'deeppink', 'gold','chocolate','y','darksalmon']
    for X in listeX:
        classe_de_X=indice_plus_proche(X,centres)
        plt.plot([X[0]],[X[1]], marker = 'o', color = formes[classe_de_X])
    for k in range(len(centres)):
        plt.plot(centres[k][0],centres[k][1],'r^')
    plt.show()
```

puis on représente, en prenant garde à bien créer des `np.array` pour le calcul des centres (par exemple avec $k = 6$) :

```
liste = [np.array([longitude[k],latitude[k]]) for k in range(len(latitude))]
classes = kmoyennes(liste,6)
dessine_donnees_avec_moyennnes(liste,classes)
```

On observe que le découpage ressemble à la découpe des continents habituelle. Plusieurs éléments mentionnés en cours sont visibles comme la dépendance aux conditions initiales pour les centres.