Correction du DS4

Problème 1 : Partie I

1. Une liste est mutable, donc non hashable, et ne peut donc pas être utilisée comme clef d'un dictionnaire.
2. Il faut juste penser à transformer les entiers de `G` en chaînes de caractères pour les concaténer

```
def position(G) :
    ch = ''
    for L in G :
        for x in L :
            ch += str(x)
    return ch
```

3. L'entier de la case (i, j) est l'élément d'indice $3 * i + j$ de `C`

```
def grille(C) :
    L = [[] for _ in range(3)]
    for i in range(3) :
        for j in range(3) :
            L[i].append(int(C[3*i+j]))
    return L
```

4. On repasse par la grille pour plus de simplicité

```
def libres(C) :
    G = grille(C)
    L = []
    for i in range(3) :
        for j in range(3) :
            if G[i][j] == 0 :
                L.append((i,j))
    return L
```

5. Comme J_1 commence et que les joueurs remplissent une seule case à tour de rôle, J_1 joue quand le nombre de cases libres est impair (il y en a 9 au départ)

```
def joueur(s) :
    if len(libres(s))%2 == 0 :
        return 2
    else :
        return 1
```

Partie II

1. Pour jouer de façon optimale, il faut commencer par ne pas perdre...La grille à évaluer est

1	1	2
	2	

il y a 5 cases libres donc c'est à J_1 de jouer, on cherche donc le maximum des étiquettes des grilles descendantes. Si J_1 ne jouent pas en (2,0), J_2 gagne au coup d'après (en jouant de manière optimale) donc toutes les grilles autres que `'112020100'` ont une étiquette égale à -1. On a donc $\mathcal{E}('112020000') = \mathcal{E}('112020100')$ et on arrive à

1	1	2
	2	
1		

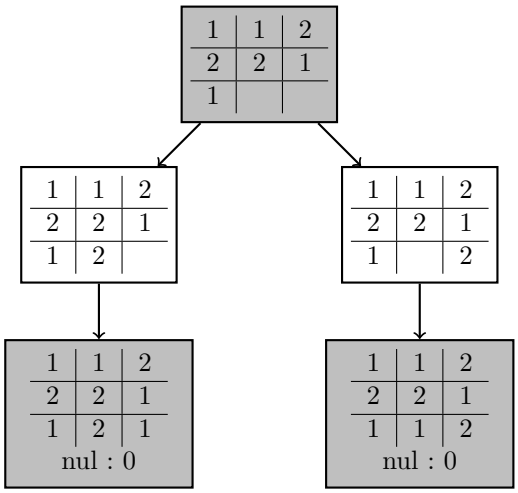
Pour les mêmes raisons, J_2 doit alors jouer en (1,0) pour ne pas perdre donc $\mathcal{E}('112020000') = \mathcal{E}('112020100') = \mathcal{E}('112220100')$ ce qui amène à

1	1	2
2	2	
1		

J_1 doit alors jouer en $(1, 2)$ pour ne pas perdre : $\mathcal{E}('112020000') = \mathcal{E}('112020100') = \mathcal{E}('112220100') = \mathcal{E}('112221100')$

1	1	2
2	2	1
1		

À partir de là rien n'est imposé donc on peut examiner le graphe restant :



Et en remontant, on trouve $\boxed{\mathcal{E}('112020000') = 0}$

2. On adapte la fonction vue en cours en rajoutant les étiquettes 0 sur les sommets sans descendants (grille complète) et sans vainqueur

```
def minmax(A,s) :
    if gagnant(s) == 1 :
        return +1
    elif gagnant(s) == 2 :
        return -1
    elif A[s] == [] :
        return 0
    else :
        j = joueur(s)
        if j == 1 :
            return max([minmax(A,f) for f in A[s]])
        else :
            return min([minmax(A,f) for f in A[s]])
```

3. Elle renvoie un dictionnaire dont les clefs sont les positions et les valeurs leur étiquette. Elle est inefficace car elle recalcule de très nombreuses fois les étiquettes des sous « arbres » : pour l'étiquette de `'000000000'`, on calcule celle de `'100000000'`, sans la mémoriser, donc on refera le calcul au moment d'entrer `'100000000'` dans le dictionnaire.

4. On modifie la fonction `minmax` pour qu'elle mette à jour le dictionnaire et on « l'enferme » dans la fonction demandée

```
def MinMax(A,s) :
    dico = {}
    def minimax(A,s) :
        if s not in dico :
            if gagnant(s) == 1 :
                res = +1
            elif gagnant(s) == 2 :
                res = -1
            elif A[s] == [] :
                res = 0
            else :
                j = joueur(s)
                if j == 1 :
                    res = max([minimax(A,f) for f in A[s]])
                else :
                    res = min([minimax(A,f) for f in A[s]])
            dico[s] = res
        return dico[s]
    minimax(A,s)
    return dico
```

Il faut exécuter `MinMax(A,'000000000')`

5. Comme on joue à la place de J_2, il faut aller vers les positions d'étiquettes minimales, donc une position qui a la même étiquette que la position actuelle

```
def jeu(s) :
    dico = MinMax(A,s)
    e = dico[s]
    for f in A[s] :
        if e == dico[f] :
            return f
```

Partie III

1. On complète les cases libres, plus facile avec la grille, en faisant attention au joueur qui doit jouer ce coup

```
def successeurs(C) :
    L = []
    k = joueur(C)
    for (i,j) in libres(C) :
        M = grille(C)
        M[i][j] = k
        L.append(position(M))
    return L
```

2. On remplit le dictionnaire récursivement ; `descendants` remplit le sous graphe des descendants de la position `C`

```
def graphe() :
    A = {}
    racine = '000000000'
    def descendants(C) :
        L = successeurs(C)
        if gagnant(C) > 0 : # partie terminée
            A[C] = []
        else :
            A[C] = L
            if L != [] :
                for s in L :
                    descendants(s)
    descendants(racine)
    return A
```

Problème 2 : Partie I

1. La complexité est $O(n)$ car le coût du test `l not in dico` est $O(1)$.

```
def lettres(mot) :
    dico = {}
    for l in mot :
        if l not in dico :
            dico[l] = 1
        else :
            dico[l] += 1
    return dico
```

2. La création des deux dictionnaires est $O(n)$ et $O(m)$; les boucles `for` aussi, la complexité totale est $O(\max\{m,n\})$

```
def Jaccard(m1,m2) :
    dico1,dico2 = lettres(m1),lettres(m2)
    d = 0
    for l in dico1 :
        if l in dico2 :
            d += abs(dico1[l]-dico2[l])
        else :
            d += dico1[l]
    for l in dico2 :
        if l not in dico1 :
            d += dico2[l]
    return d
```

3. Le calcul de $d_{i,j}$ passe par le calcul (optimal) de sous-problèmes donc on a la propriété de sous structure optimale
4. On remplit un tableau de taille $n_1 \times n_2$ (donc les coût sont $O(n_1 n_2)$) et on renvoie la dernière valeur du tableau

```
def lev(ch1,ch2) :
    n1,n2 = len(ch1),len(ch2)
    T = [[i+j for j in range(n2+1)] for i in range(n1+1)]
    for i in range(1,n1+1) :
        for j in range(1,n2+1) :
            if ch1[i-1] == ch2[j-1] :
                T[i][j] = T[i-1][j-1]
            else :
                T[i][j] = 1 + min(T[i][j-1],T[i-1][j],T[i-1][j-1])
    return T[-1][-1]
```

Partie II

1. C'est la même fonction que `lettres` (sauf qu'elle s'applique à une liste)

```
def occurrences(L) :
    dico = {}
    for x in L :
        if x not in dico :
            dico[x] = 1
        else :
            dico[x] += 1
    return dico
```

2. Comme on sait qu'on cherche des entiers naturels, on peut commencer les comparaisons avec 0

```
def majoritaire(L) :
    M = 0
    dico = occurrences(L)
    for x in dico :
        if dico[x] > M :
            r,M = x,dico[x]
    return r
```

3. On commence par créer une liste de couples dont la première coordonnée est la distance aux point de `Dico` et la seconde est la langue du point de `Dico`

```
def kNN(x,Dico,k) :
    L = [(d(x,y),Dico[y]) for y in Dico]
    L.sort()
    Langues = [L[1] for i in range(k)]
    return majoritaire(Langues)
```

4. On utilise `D` pour vérifier si nos prédictions sont exactes

```
def precision(k,D) :
    X,E = couper(D)
    exacte = 0
    for x in X :
        predite = kNN(x,E,k)
        if predite == X[x] :
            exacte += 1
    return exacte/len(X)
```

5. Le choix de k est primordial dans l'utilisation de cet algorithme. Le fait de couper `langues` en deux permet de faire des tests sur une moitié de ce dictionnaire (dont on connaît les vraies langues) donc de vérifier la précision des prédictions et donc d'ajuster au mieux la valeur de k pour faire ensuite des prédictions sur des mots inconnus.
6. Il suffit de rajouter toutes les langues correspondant à chaque mot (ne retenir que la première ne serait pas judicieux car on risquerait de ne pas retenir la bonne), par concaténation de listes. Cette fois la seconde coordonnée des couples de `L` est une liste de langues et la liste `Langues` aura au final plus de k éléments

```
def kNN1(x,Dico,k) :
    L = [(d(x,y),Dico[y]) for y in Dico]
    L.sort()
```

```
    Langues = []
    for l in L[1] :
        Langues += l
    return majoritaire(Langues)
```