Interrogation d'informatique n°6. Corrigé

Exercice A. Partitionnement optimal en dimension 1

1)

```
def min(L) :
    m = L[0] ; n = len(L)
    for j in range(1,n) :
        if L[j] < m :  m = L[j]
    return m
```

2)

```
def moyenne(E,i,j) :
    mu = 0
    for k in range(i,j) :  mu += E[k]
    return mu/(j-i)
```

3)

```
def distance(E,i,j) :
    d = 0
    mu = moyenne(E,i,j)
    for k in range(i,j) :  d += (E[k] - mu) ** 2
    return d
```

4) Il faut faire attention à la manière dont sont délimitées les classes. On traite la dernière à part.

```
def score(E, P):
    N,K = len(E),len(P)
    S = distance(E,P[K-1],N)
    for i in range(K-1) :  S += distance(E,P[i],P[i+1])
    return S
```

5)

```
def classes_plus_proches(E,P) :
    N,K = len(E),len(P)
    Lmu = [0]*K
    for i in range(K-1) :  Lmu[i] = moyenne(E,P[i],P[i+1])
    Lmu[K-1] = moyenne(E,P[K-1],N)
    iopt = 0
    for i in range(K-1) :
        if Lmu[i+1] - Lmu[i] < Lmu[iopt+1] - Lmu[iopt] :  iopt = i
    return iopt
```

6) On crée une nouvelle liste de taille $K-1$, qu'on remplit en fonction de iopt. Il s'agit juste de supprimer l'élément d'indice $i_{opt}+1$ (mais on crée ici une nouvelle liste pour ne pas modifier l'ancienne).

```
def fusion_classes(P,iopt) :
    K = len(P)
    nouv_P = [0]*(K-1)
    for i in range(K-1) :
        if i <= iopt :  nouv_P[i] = P[i]
        else :  nouv_P[i] = P[i + 1]
    return nouv_P
```

7)

```
def CHA(E, K):
    N = len(P)
    P = [i for i in range(N)]
    while len(P)>K :
        iopt = classes_plus_proches(E, P)
        P = fusion_classes(P, iopt)
    return P
```

8) Lorsque $k=1$, $s(n,k)$ vaut $D(0,n)$ car il n'y a qu'une seule classe. Ce score peut être calculé en $O(n)$.

9) Une partition des n éléments de taille k consiste à choisir une dernière classe $\{x_i,...,x_{n-1}\}$ et une partition des i premiers éléments en $(k-1)$ classes. Comme aucune classe ne doit être vide, on a nécessairement $i \geq k-1$. Pour obtenir une partition optimale, il faut trouver une sous-partition optimale des i premiers éléments, puis minimiser les scores obtenus sur toutes les valeurs de i possibles.

10) On écrit une fonction auxiliaire `aux(E,n,k,dico)` qui prend en argument E, des
entiers n et k et un dictionnaire mémoïsant les résultats et renvoie `s(n,k)`. Une fois cette fonction écrite, il suffit de lancer un appel avec $n=N$ et $k=K$, en utilisant un dictionnaire initialement vide.

```
def aux(E,n,k,dico) :
    if (n,k) not in dico :
        if k == 1 :
            dico[(n,k)] = distance(E,0,n)
        else :
            L = [aux(E,i,k-1,dico)+distance(E,i,n) for i in range(k-1,n)]
            dico[(n,k)] = min(L)
    return dico[(n,k)]

def clustering_dynamique(E,K) :
    dico = {} ; return D(E,len(E),K,dico)
```

11) Il y a $O(N \times K)$ valeurs qui sont calculées dans le dictionnaire.

De plus, chaque valeur nécessaire de calculer le minimum par la boucle `for`. Cette boucle, de taille $n-k$, fait un appel à distance de complexité $O(n-i)$. On obtient une complexité totale en $O(K \times N^3)$.

Exercice B. Noyau et numérotation 0-1 de Sprague-Grundy

1)

```
def sansSuccesseur(d) :
    for s in d :
        if d[s] == [] :  return s
    assert True , "graphe non acyclique"
```

2)

```
def transpose(d) :
    dT = {}
    for s in d :  dT[s] = []
    for s in d :
        for t in d[s] :  dT[t].append(s)
    return dT
```

3)

```
def supprimeSommet(d,s) :
    dT = transpose(d) ; L = dT[s] ; L.append(s)
    dNew = {}
    for u in d :
        if not (u in L) :  # on ne considère que les sommets non présents dans L
        dNew[u] = []
        for v in d[u] :
            if not (v in L) : dNew[u].append(v)
    return dNew
```

4)

```
def noyau(d) :
    N = []
    while len(d)>0 :
        s = SansSuccesseur(d) ; N.append(s)
        d = supprimeSommet(d,s)
    return N
```

5)

```
def retrait(L,j) :
    L = L.copy() ; k = j
```

```
    # on commence par déterminer le plus grand entier k tel que L[k] = L[j]
    while (k+1 < n) and L[k+1] == L[k] :  k = k+1
    # il ne reste plus qu'à décrémenter L[k] de 1 (et à le supprimer s'il vaut 1)
    if L[k] == 1 :  return L.pop()
    L[k] = L[k] - 1 ; return L
```

6)

```
def successeurs(L) :
    liste = [] ; n = len(L)
    for j in range(n) :
        M = L
        for _ in range(L[j]) :
            M = retrait(M,j) ; liste.append(M)
    return liste
```

7) La clé ne pouvant être un objet mutable, on convertit la liste associée en une chaîne de caractère.

```
def marienbad(L) :
    d = {} ; pile = [L]
    while pile :
        M = pile.pop()
        if (not (str(M) in d)) :
            liste = successeurs(M)
            d[str(M)] = []
            for N in liste :      # le cas où N est la liste vide n'est pas une position
              if N :
                  pile.append(N) ; d[str(M)].append(N)
    return d
```

8) a) Le joueur A choisit une stratégie δ telle que $\forall x \in R$, $\delta(x) \in G[x] \cap N$.

Remarque : Une telle stratégie existe car tout élément de R admet au moins un sucesseur dans N.

Supposons que le joueur A se trouve en $x \in R$. Il joue selon la stratégie choisie et donc le joueur B se retrouve en une position $y \in N$. On a aors deux cas ;

- si $y \in T$, le joueur B a perdu.
- sinon, le joueur B joue, mais comme tous les successeurs de y appartiennent à R, le coup qu'il joue ramène le joueur A en une position $x' \in R$.

On obtient ainsi une suite de positions alternativement dans R (pour le joueur A) et dans N (pour le joueur B). La suite termine car le graphe est acyclique. La dernière position appartient à T, donc à N, donc c'est au joueur B de jouer, et ainsi il a perdu.

b) Il suffit d'inverser les rôles des joueurs A et B. Si B choisit la bonne stratégie, le joueur A va rester dans N dès l'instant où

il y entre, et il est alors certain de perdre.

9)

```
def sprague(d) :
    dS = {}
    def traite(s) :
        if not(s in dS) :
            dS[s] = 0
            for t in d[s] :
                traite(t)
                if dS[t] == 0 :  dS[s] = 1
    for s in d :  traite(s)
    return dS
```

10) (*existence*) On considère $N = \{s \in S \mid f(s) = 0\}$. Montrons que N est stable et absorbante.

- Si $s \in N$, c'est-à-dire $f(s) = 0$, aucun successeur t de s ne vérifie $f(t) = 0$. Donc N est stable.

- Si $s \notin N$, c'est-à-dire $f(s) = 1$, alors s admet au moins un successeur t vérifiant $f(t) = 0$, c'est-à-dire s admet au moins un successeur dans N, donc N est absorbant.

(*unicité*) Considérons une partie M stable et absorbante. On vérifie par induction (c'est-à-dire par exemple récurrence sur la profondeur) que $s \in M$ ssi $f(s) = 0$.

En effet :

- Si s est un sommet sans successeur, alors on a nécessairement $s \in M$ (car M est stable) et on a aussi $f(s) = 0$.

- Soit un sommet s. Supposons par hypothèse de récurrence que pour tout successeur t de s, on a : $t \in M$ ssi $f(s) = 0$. Alors $s \in M$ ssi tous les successeurs t vérifient $f(t) = 1$, donc ssi $f(s) = 0$.

Ainsi, si M est stable et absorbante, alors $M = \{s \in S \mid f(s) = 0\}$.

Variante : on suppose par l'absurde qu'il existe deux nodistincts N_1 et N_2. On pose $M = N_1 \Delta N_2$ les éléments qui appartiennent qui parrtiennent à l'un et pas à l'autre. On vérifie aisément que si $x \in M$, alors x admet un successeur dans M. Ce qui en itérant conduit (par le principe des tiroirs à un cycle dans le graphe G, ce qui est absurde.