TP 2 Partie 1

Le but de ce TP est de revoir les graphes, se familiariser avec leur représentation sous forme de dictionnaire (liste d'adjacence) ou leur matrice d'adjacence, et aussi de mettre en application les techniques de hachages sur les dictionnaire.

1 Crypter un texte

Dans cette première partie nous allons travailler sur un algorithme de cryptage, assez proche du hachage.

Exercice 1.1 *On cherche à crypter un texte* `t` *de longueur* n*, qui sera une liste de* n *entiers compris entre* 0 *et* 25 *(correspondant au 26 lettres de l'alphabet).*

1. *Proposer une fonction* `codage(t,d)` *renvoyant une liste de même taille que* `t` *décaler de* `d` *modulo* 26*. Puis une fonction décodage renvoyant la liste d'origine.*
2. *On part du principe que dans un texte crypter (en français) la lettre e apparait plus fréquement. Proposer une fonction* `frequence(t')` *renvoyant une liste de taille* 26 *dont la case* i *indique le nombre d'apparition de l'indice* i*.*
3. *On suppose le texte crypter avec un décalage* d *inconnu, proposer une fonction* `decodageauto(t)`*, appelant les précédente et proposant un décodage de* t*.*
4. *On améliore le cryptage en proposant un décalage different pour chaque lettre. On se donne une clé, par exemple,* `concours` *c-a-d* $[3, 15, 14, 3, 15, 20, 18, 19]$*, qui transforme a en c, puis a en o et ainsi de suite. Donc le premier nombre est décaler de* 3 *modulo 26 puis* 15 *modulo 26 et ainsi de suite jusqu'au huitième, puis on recommence*
   (a) *Proposer la fonction* `codageV(t,c)` *prenant la liste* t*, la clé (sous forme de liste) et renvoyant le texte codé.*
   (b) *Programmer la fonction* `pgcd(a,b)` *renvoyant le pgcd des entiers* a *et* b*.*
   (c) *On suppose un texte* t *(assez long) codé par* t' *et on veut retrouver la clé. problème une même lettres dans* t' *n'est pas forcément la même pour* t*. On va donc rechercher la répétition, non pas d'une lettre, mais de* 3 *lettres et si cette répétition est suffisamment grande (on a des multiples de* k *la longueur de la clé), le pgcd donne la longueur de la clé.*

      Programmer une fonction `pgcdDrep(t',i)`*, prenant comme paramètre le texte crypter* t' *de taille* n*,* i *un indice,* $(0 \leq i \leq n-2$ *) et qui renvoie le pgcd de toutes les distances entre les répétition (dans* t'*) de la séquence* `t'[i],t'[i+1],t'[i+2]` *dans le texte* `t[i+2 :]`*,* 0 *si pas de répétition..*
   (d) *Programmer* `longueur Cle(t')` *qui renvoie la longueur de la clé (on pourra déterminer la complexité en fonction de* n *de ce programme).*
   (e) *Proposer une fonction de décodage.*

Exercice 1.2 *Pours les* $\frac{5}{2}$

En base 3*, les entiers* $0, 1, 2, 3, 4, 5, 6, 7, 8$ *sont représentés par* $00, 01, 02, 10, 11, 12, 20, 21, 22$*. Le chiffre de poids fort de bc est b ; le chiffre de poids faible est c.*

1. (a) *Proposer une fonction* `poids(bc)` *prenant comme paramètre le chiffre bc de poids respectifs b et c et renvoyant le uplet* `(b,c)`*.*
   (b) *Écrire la fonction* `entier(b,c)` *renvoyant l'entier compris entre* 0 *et* 8 *qui s'écrit bc en base* 3*.*
   (c) *Soit* x *un entier vérifiant* $0 \leq x \leq 8$*. Ècrire une fonction* `poidsFort(x)` *retournant le chiffre de poids fort de* `x` *en base* 3*.*

(d) *Écrire la fonction* `poidsFaible(x)` *retournant le chiffre de poids faible de* `x`.

2. *Dans ce problème, les textes sont représentés en représentation ternaire. Un savant russe nous a convaincus de la pertinence de ce choix plus compact que la représentation binaire. Un texte est rangé dans un tableau* `t` *(une liste de liste) de* N *caractères vérifiant* `t[i]` *est un entier appartenant à* $\{0,1,2\}$ *pour tout* i *vérifiant* $0 \leq i < N-1$ *; par ailleurs* `t[N-1]`$= X > 2$ *(le dernier caractère n'est pas ternaire). On suppose* $N \geq 1$.

Quelques définitions sont nécessaires :

— *la chaîne de caractères de longueur* l *démarrant en* i *est la liste* `[t[i],t[i+1],...t[i+l-1]]`.

— *On dira que deux chaînes* `[t[i],t[i+1],...t[i+l-1]]` *et* `[t[j],t[j+1],...t[j+l'-1]]` *sont égales si* $l = l'$ *et* $t[i+k] = t[j+k]$ *pour* $0 \leq k < l$.

(a) *Proposer une fonction* `Conversion(t)` *prenant une chaine d'entiers* `t` *compris entre* 0 *et* 8 *et renvoyant la une chaine de nombre converti en ternaire.*

(b) *Après avoir importer la bibliothèque* `numpy.random`*, tester la fonction conversion sur une chaine bien choisit de longueur* 10.

(c) *Écrire une fonction* `longueurMotif(t,i,j,m)` *qui retourne, avec une complexité linéaire, c-a-d d'ordre* $O(N)$*, la plus grande longueur* l *d'une chaîne de* `t` *démarrant en* i *égale à une autre chaîne de* `t` *démarrant en* j*. En outre, cette longueur doit vérifier* $l \leq m$ *(on prendra garde de ne pas dépasser la longueur de la chaine* `t`*...). Tester le programme sur une chaine de taille* 50.

(d) *On suppose* $i < j$*. Écrire une fonction* `longueurMotifMax(t, i, j, m)` *qui retourne, avec une complxité quadratique, c-a-d* $O(N^2)$ *, la plus grande longueur* l *d'une chaîne démarrant en* $i+k$ *égale à une chaîne démarrant en* j *pour* $0 \leq k < m$*. En outre, on exige* $i + k < j$ *et* $l \leq m$*. Tester le programme sur une chaine de taille* 50.

(e) *Modifier la fonction précédente pour obtenir la fonction* `motifMax(t,i,j,m)` *qui renvoie les variable global* `A,L,C`*, avec une complexité quadratique :* `L` *la plus grande longueur* l *d'une chaîne démarrant en* $i+k$ *égale à une chaîne démarrant en* j *pour* $0 \leq k < m$ *;* `A` *la valeur de* k *pour lequel* $i+k$ *est l'indice de départ de cette chaîne de longueur maximale ;* `C` *le caractère suivant cette chaîne à partir de* j *dans* t*. À nouveau, cette longueur doit vérifier* $l \leq m$*. Et on a* $i+k < j$*. On pourra compléter :*

```
def motifmax(t,i,j,m):
    global A,L,C
    ..................
    return A,L,C
```

3. *La méthode de compression de Ziv et Lempel, adoptée dans les commandes zip ou gzip, consiste à repérer les motifs maximaux déjà rencontrés dans un texte et à indiquer pour chacun d'eux le triplet* `(A,L,C)` *calculé dans la question précédente entre toute paire d'indices* i *et* j*. Pour mesurer le facteur de compression, nous utilisons le même codage pour ces triplets que pour les caractères du texte, c'est-à-dire le système ternaire dans ce problème.*

(a) *Écrire une fonction* `imprimerTriplet(A,L,C)` *qui renvoie les arguments* `A, L, C` *sous forme de cinq chiffres consécutifs, les deux caractères ternaires de* `A`*, puis les deux caractères ternaires de* `L`*, puis le chiffre* `C`*, en imposant* $0 \leq A < 9$*,* $0 \leq L < 9$ *et* $0 \leq C \leq 9$.

(b) *On suppose à présent que* `t` *contient un long texte ternaire commençant par* 9 *caractères* `0` *; en outre,* `t` *finit par un caractère* `x` *spécial (*`x` > 2*). On déplace sur ce texte une fenêtre de longueur* 18*. Au début, cette fenêtre est alignée à gauche sur le début du tableau, et on pose* $j = 9$*. En régime de croisière, la dixième case de la fenêtre correspond à l'entrée* j *du tableau* `t`*. On recherche, dans la partie gauche de la fenêtre, la chaîne de longueur* l *maximale vérifiant* $l < 9$ *et égale à une chaîne de caractères démarrant en* j*. On imprime, grâce à la fonction* `imprimerTriplet`*, le triplet* `(A, L, C )` *donné par* `motifMax`*. Puis, on recentre la fenêtre sur le caractère suivant le caractère d'arrêt* `C`*. Ce processus continue jusqu'au bout du tableau* `t` *comme indiqué par la figure. Ainsi pour le texte suivant, on obtient les décompositions de chacune des lignes, soit au total le facteur de compression* 30/37 *qui serait nettement meilleur dans une base supérieure à* 3 *et si la taille de la fenêtre était plus grande que* 18 *(Il y a en effet* 30 *caractères dans le résultat et* 37 *dans le texte d'entrée). Écrire une fonction* `compresser(t)` *qui renvoie le texte compressé par la méthode précédente.*

(c) *Pour la décompression, on produit d'abord* 9 *caractères* 0. *On considère ensuite tous les triplets* `(A, L, C )` *représentés par* 5 *caractères ternaires consécutifs et on recrée la chaîne originale jusqu'au dernier triplet dont la composante* `C` *n'est pas comprise entre* 0 *et* 2. . *Écrire une fonction* `décompresser(tc)` *qui prend un texte ternaire* `tc` *comme paramètre et qui renvoie le texte décompressé correspondant.*

2 Les arbres

Un arbre pondéré est un graphe auquel on affecte une valeur (poids) à une arrête, le poids d'un chemin est alors la somme des pondérations des arrêtes. On peut alors représenter l'arbre soit comme une matrice A en affectant la pondération de l'arrête reliant le noeud i au noeud j au coefficient $A_{i,j}$ et $+\infty$ si les noeuds ne sont pas reliés. Ou choisir de le représenter avec une liste d'adjacance en choisissant le couple $\{u_i : [(v_1, p_1), \ldots, (v_k, p_k))\}$ ou v_j est un noeud et p_j le poids de l'arrête reliant le noeud u_i au noeud v_j.

Exercice 2.1 *Proposer une fonction* `listadj(A)` *prenant comme paramètre la matrice d'adjacence d'un graphe (on pourra supposer que A n'est composée que d'entier* 0 *ou* 1, *donc que le graphe n'est pas pondéré) et renvoyant un dictionnaire (ou une liste) de liste d'adjacence.*

Des commandes de tris :

```
L.sort() # tri par ordre croissant
L.reverse() #  change l'ordre
min(L) # ressort le min
max(a) # je vous laisse deviner
L.insert(i,a)# insert le terme a en position i
del L[i] # efface le terme en position i
L[i:j] # extrait la sous liste d'indice i a j-1
L.remove(a)# retirer l'element a de L
L.count(i)# indique le nombre de terme de L de valeur i
```

Définition 2.1 *Colorer un graphe connexe non orienté (sans boucle c-a-d aucune arrete* (u, u)*), c'est associer une couleur à chaque noeud de telle sorte que deux noeuds adjacents soient de couleur différentes. Le nombre chromatique d'un graphe est le nombre minimal nécessaire de couleurs pour colorer un graphe. Il est inférieur à* $\Delta + 1$ *où* Δ *est le degré maximum des noeuds du graphe.*

On propose deux algorithmes permettant d'obtenir une coloration d'un graphe et donc une "idée" très approchée de ce nombre chromatique :

Exercice 2.2 *Algorithme Welsh Powell*

Etape 1 *Ranger les sommets dans l'ordre décroissant de leur degré*

Etape 2 *Choisir une nouvelle couleur dite courante et colorer ainsi le premier sommet de la liste non coloré*

Etape 3 *Dans la liste des sommets, colorer avec la couleur courante, les sommets de la liste non encore colorés qui ne sont pas adjacents avec un noeud coloré avec cette couleur.*

Etape 4 *Si tous les sommets ne sont pas encore colorés revenir à l'étape 2*

1. *Programmer la fonction* `deg(A,i)` *prenant pour paramètre la matrice d'adjacence A qui n'a que des* 1 *et des* 0, *et retournant un entier correspondant au degré du noeud i (on pourra appeller la fonction* `listadj()`*).*
2. *Programmer la fonction* `listeNoeud_degre(A)` *retournant une liste de liste* `[deg(n),n]` *du couple n, le numéro du noeud, et deg*(n) *son degré.*
3. *Programmer la fonction* `ordonnernoeud(A)` *prenant pour paramètre la matrice d'adjacence du graphe et retournant la liste des noeuds ordonner dans l'ordre décroissant de leur degré.*

4. *On défini* `couleur` *le dictionnaire où* `couleur[i]` *est la couleur du noeud i (les clés sont donc les entiers de 1 à n), est un entier non nul s'il est coloré. Programmer la fonction* `nbcouleur(couleur)` *prenant comme paramètre le dictionnaire* `couleur` *et renvoyant le nombre de couleur c différentes..*
5. *Proposer une fonction prenant pour paramètre la matrice d'adjacence d'un graphe et retournant l'entier c indiquant le nombre de couleurs utilisées par l'algorithme de Welsh Powell et la coloration* `couleur` *en complétant ce qui suit :*

```
def welshpowell(A):
    L=ordonnernoeud(A)
    adj=listadj(A)
    n=len(A)
    couleur={} # la i:j indique noeud i de couleur j
    c=0
    while len(couleur)<n:

        .........
    return c , couleur
```

Ce premier algorithme n'est pas franchement folichon, on en propose un autre.

On considère un graphe $G = (V, E)$ simple connexe et non-orienté. Pour chaque sommet v de V, on calcule le degré de saturation $DSAT(v)$ de la manière suivante :

$DSAT(v)$ = **nombre de couleurs différentes dans les sommets adjacents à v.** A est toujours la matrice d'ajacence composée de 0 et 1 du graphe G sans boucles, connexe et non orienté.

Exercice 2.3 *L'algorithme* `DSATUR` *est un algorithme de coloration séquentiel, au sens où il colore un seul sommet à la fois et tel que :*

Etape 1 *Au départ le graphe n'est pas coloré.*

Etape 2 *Ordonner les sommets par ordre décroissant de degré.*

Etape 3 *On colore un des sommets de degré maximum avec la couleur* 1*.*

Etape 4 — *Choisir un sommet non coloré avec DSAT maximum. En cas d'égalité, choisir un sommet de degré maximal.*
— *Colorer ce sommet par la plus petite couleur possible.*

Etape 4 *On stoppe l'algorithme* `DSATUR` *quand tous les sommets de G sont colorés sinon retour à l'étape 4.*

1. *Programmer la fonction* `DSAT(A,couleur,i)` *prenant pour paramètre la matrice d'ajacence A la liste des couleurs des noeuds du graphe et l'entier i et retournant le DSAT(i).*
On pourra compléter le programme suivant :

```
def DSATUR(A,couleur,i):
    adj=[]
    for j in range(len(A[i])):
        if A[i][j]!=0:
            adj.append(j)
    dsatur=0
    ..............
    ..............
    ..............
    return dsatur
```

Que représente la liste `adj` *?*

2. *On se donne les deux fonctions suivante :*

```
def deg(A,n):
        return sum(A[n])
```

```
def ordonnernoeud(A):
    L=[]
    for k in range(len(A)):
          L.append([deg(A,k),k])
    L.sort(reverse=True)
    return [ u[1] for u in L]
```

3. *On se donne la liste couleur tel que* `couleur[i]` *renvoie la couleur (un entier) du noeud i. Par convention un noeud de couleur nulle sera sans couleur.*

   (a) *Proposer une fonction* `DSAT_max(L,couleur,A)` *qui prend comme paramètre une liste de noeuds ordonner en degré décroissant, la liste couleur, la matrice d'adjacence et qui retourne le noeud de* `L` *de DSATUR maximum.*

   (b) *Proposer une fonction* `couleur_min(couleur,A,j)` *qui prend comme paramètre, la matrice d'adjacence, la liste couleur et un noeud i et qui renvoie un entier non nul, le plus, petit possible correspondant à la couleur du noeud i (en faisant en sorte que aucun noeud adjacent à i soit de même couleur)*

   (c) *Proposer un programme, découpé en plusieurs fonctions permettant de retourner le nombre de couleur nécéssaire pour la coloration DSATUR. En complétant :*

```
def colorDSATUR(A):
    noeud=ordonnernoeud(A)
    n=len(A)
    couleur=[0]*n
    ..........................
    return couleur
```