Devoir ITC n°2

Réduction d'images

I. Introduction

L'objectif de ce sujet est de réduire la largeur d'une image.

Le terme **pixel** provient de l'abréviation anglaise de *picture element* qui signifie élément d'image. Un pixel est l'unité de base qui permet de mesurer la définition d'une image numérique.

Une image numérique est un tableau (matrice) de n lignes et p colonnes dont les coefficients sont les valeurs des pixels. Dans le cas des images numériques en niveaux de gris, la valeur du pixel est généralement enregistrée dans l'ordinateur ou l'appareil photo numérique sous forme de nombres entiers entre 0 et 255 soit 256 valeurs possibles pour chaque pixel. La valeur 0 est attribuée au noir, et 255 correspond au blanc.

En fait, les nombres entiers sont stockés en écriture binaire, une succession de 0 et de 1, eux-mêmes enregistrés sur une unité élémentaire de stockage appelée *bit*. Les $256 = 2^8$ valeurs possibles des pixels sont donc codées sous 8 bits, donc un octet.

Dans ce sujet une image en niveau de gris est codée par une matrice I d'entiers compris entre 0 (pour noir) et 255 (pour blanc). La hauteur de l'image correspond au nombre de lignes n de la matrice et sa largeur est donnée par le nombre de colonnes p de I.

La matrice est représentée par la liste `image` . Il s'agit d'une liste de listes correspondant aux lignes de l'image (listes d'entiers). Ainsi pour $i \in [\![0; n-1]\!]$ et $j \in [\![0; p-1]\!]$ le pixel en position (i, j) de l'image noté $I_{i,j}$ correspond en Python à `I[i][j]` .

Figure 1 : Image libre[1]

1 – Quel est le nombre de colonnes d'une image carrée (bitmap) dont l'espace mémoire est 360 ko (sans compression) ?

---

360 000 octets (1 octets = 8 bits : code 1 pixel) correspond à 600×600.

---

2 – Écrire une fonction `dim(image:list)->tuple` prenant en argument une image et qui retourne les deux valeurs n et p où n est la hauteur et p la largeur de l'image.

```
def dim(image:list) -> tuple:
    nb_ligne = len(image)
    nb_colonne = len(image[0])
    return nb_ligne, nb_colonne
```

---

[1] Source : *https://pixabay.com/photos/birds-perch-perched-africa-avian-4870045/*

3 – Rédiger une fonction d'entête `nulle(nb_ligne, nb_colonne)->list` qui retourne une liste de listes qui correspond à la matrice nulle de `nb_ligne` lignes et `nb_colonne` colonnes. Il s'agit d'une image complètement noire.

```
def nulle(nb_ligne, nb_colonne):
    return [[0 for j in range(nb_colonne)] for i in range(nb_ligne)]
```

4 – Rédiger une fonction d'entête `copie_im(image: list) -> list` qui retourne une liste de listes qui correspond à une copie indépendante de la matrice image fournie en paramètre.

```
def copie_im(image) -> list:
    return [[x for x in ligne] for ligne in image]
```

Une image couleur est en fait la réunion de trois images (ou canaux) : une rouge, une verte et une bleue. Cette représentation s'apparente au fonctionnement du système visuel de l'homme. Chaque pixel est alors représenté par un triplet (*Red, Green, Blue),* chacun étant un entier codé sur 8 bits, on dispose à présent d'un codage sur 24 bits soit plus de 16 millions de couleurs.
Par exemple : (255, 0, 0) correspond à un pixel rouge, (0, 255, 0) représente un pixel vert, (0, 0, 255) est le pixel bleu. Par additivité (255, 0, 255) devient le pixel magenta.

Une image couleur est donc une matrice dont les éléments sont des listes de 3 entiers : `[Red, Green, Blue]` . Pour travailler en nuances de gris, on peut utiliser la recommandation 709 de la Commission Internationale de l'Éclairage selon laquelle

$$Grey = 0.2125 \times Red + 0.7154 \times Green + 0.0721 \times Blue$$

5 – Proposer une fonction `rgb2gray(image:list)->list` qui renvoie une matrice de même dimension mais dont les pixels sont *Gris*. L'argument `image` est une matrice « couleur » dont les éléments (pixels) sont des listes `[Red, Green, Blue]` de trois entiers. La conversion utilise la recommandation ci-dessus.

```
def rgb2gray(image):
    nb_ligne, nb_colonne = dim(image)
    new = nulle(nb_ligne, nb_colonne)
    for i in range(nb_ligne):
        for j in range(nb_colonne):
            Red, Green, Blue = image[i][j]
            new[i][j] = round(0.2125 * Red + 0.7154 * Green + 0.0721 * Blue)
    return new
```

La suite du sujet ne considère que des **images en niveaux de gris** dont les éléments (pixels) sont des entiers compris entre 0 et 255. Pour simplifier on suppose que le nombre de lignes et le nombre de colonnes sont **pairs**.

II. Redimensionnement naïf

Pour réduire la largeur de moitié une première méthode naïve consiste à ne prendre qu'un pixel sur deux dans chaque ligne de la matrice.

6 – Écrire une fonction `réduction_moitié(image:list)->list` qui prend en argument une image `image` et qui retourne une image deux fois moins large selon la première méthode naïve. Donner sa complexité en fonction de n et p.

```python
def réduction_moitié(image: list):
    nb_ligne, nb_colonne = dim(image)
    new = nulle(nb_ligne, nb_colonne//2)
    for i in range(nb_ligne):
        for j in range(0, nb_colonne, 2):
            new[i][j//2] = image[i][j]
    return new
```

Complexité : la création de l'image est en $\mathcal{O}(np)$, la double boucle ne contient que des opérations élémentaires donc en $\mathcal{O}(np/2)$, soit au total $\mathcal{O}(np)$.

---

7 – Une variante consiste à fusionner les pixels voisins en un unique pixel prenant comme valeur leur demi-somme. Écrire une fonction `réduction_demi(image: list)->list` qui prend en argument une image `image` et qui renvoie une image deux fois moins large selon cette seconde méthode naïve.

```python
def réduction_demi(image:list):
    nb_ligne, nb_colonne = dim(image)
    new = nulle(nb_ligne, nb_colonne//2)
    for i in range(nb_ligne):
        for j in range(nb_colonne//2): # variante
            new[i][j] = (image[i][2*j] + image[i][2*j+1]) // 2
    return new
```

Les images obtenues présentent le défaut principal d'être déformées comme l'exemple ci-contre.

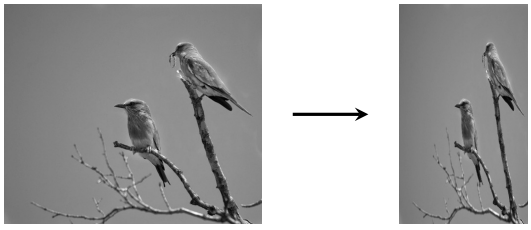

Figure 2 : Résultat de la méthode naïve

III. Nouvelle approche : l'énergie

3.1. Définition

En notant $I_{i,j}$ le niveau de gris du pixel de coordonnées (i, j), l'énergie de ce pixel intérieur à l'image est définie par :

$$e_{i,j} = \left|\frac{I_{i+1,j} - I_{i-1,j}}{2}\right| + \left|\frac{I_{i,j+1} - I_{i,j-1}}{2}\right|$$

En effet, tous les pixels n'ont pas le même poids dans une image, ainsi la suppression d'un pixel entouré de pixels de la même couleur se remarque moins que la suppression d'un pixel de contour. L'énergie est d'autant plus faible que le pixel est dans une zone uniforme de l'image, et inversement elle est d'autant plus grande qu'il est dans une zone de forte variation du niveau de gris (par exemple au niveau d'un contour). Cette quantité n'est autre qu'une discrétisation de la norme du gradient.

Les bords sont traités de façon légèrement différente. Si $i = 0$, $e_{i,j} = |I_{i,j} - I_{i+1,j}|$ à la place de $\left|\frac{I_{i+1,j} - I_{i-1,j}}{2}\right|$, si $i = n - 1$ on met $|I_{i-1,j} - I_{i,j}|$ à la place de $\left|\frac{I_{i+1,j} - I_{i-1,j}}{2}\right|$. On procède de façon analogue avec la deuxième valeur absolue si $j = 0$ ou $j = p - 1$.

8 – Proposer une fonction `énergie(image: list)` qui prend en argument une image `image` et qui retourne une image correspondant à son énergie. Les valeurs des pixels ne sont plus nécessairement des entiers.

```
def energie(image:list) -> list:
    n, p = dim(image)
    enrj = nulle(n, p)
    for i in range(n):
        for j in range(p):
            if i == 0:
                dx = image[1][j] - image[0][j]
            elif i == n - 1:
                dx = image[i][j] - image[i-1][j]
            else:
                dx = (image[i+1][j] - image[i-1][j]) / 2
            if j == 0:
                dy = image[i][1] - image[i][0]
            elif j == p - 1:
                dy = image[i][j] - image[i][j-1]
            else:
                dy = (image[i][j+1] - image[i][j-1]) / 2
            enrj[i][j] = abs(dx) + abs(dy)
    return enrj
```

3.2. Réduction ligne par ligne

Une première approche basée sur l'énergie, consiste à supprimer pour chaque ligne le pixel de plus faible énergie et de répéter l'opération.

9 – Proposer une fonction `indice_min(ligne: list)->int` qui à partir d'une liste `ligne` renvoie l'indice du plus petit élément de la liste donnée en argument.

```
def indice_min(ligne: list) -> int:
    mini = 0
    p = len(ligne)
    for j in range(p):
        if ligne[j] < ligne[mini]:
            mini = j
    return mini
```

10 – On rappelle que la méthode `pop` n'est autorisée que en dernière position . Écrire une procédure `retirer_indice(ligne: list, ind: int) -> None` qui modifie une liste fournie en retirant l'élément en position `ind` . On pourra s'inspirer du tri par insertion.

```
def retirer_indice(ligne: list, ind: int) -> None:
    p = len(ligne)
    for i in range(ind, p-1):
        ligne[i] = ligne[i+1]
    ligne.pop()
```

11 – Écrire une fonction `réduction_ligne(image: list, , nb: int) -> list` qui prend en paramètres une image et renvoie une copie de l'image dans laquelle on a retiré les `nb` pixels de plus faible énergie de chaque ligne (il s'agit donc d'une liste de taille $n \times (p - nb)$). Quelle est sa complexité ?

```
def réduction_ligne(image: list, nb: int) -> list:
    new = copie_im(image)   # O(n p))
    enrj = énergie(image)   # O(n p)
    n, p = dim(image)       # O(1)
    for répétition in range(nb):  # nb tours
        for i in range(n):        # n tours
            ind = indice_min(enrj[i])     # O(p)
            retirer_indice(new[i], ind)   # O(p)
            retirer_indice(enrj[i], ind)  # O(p)
    return new
```

Les copies et calculs de l'énergie sont en $\mathcal{O}(np)$, l'opération de retirer un pixel sur une ligne coûte $\mathcal{O}(p)$, est répétée n fois pour chaque ligne, et le tout est répété nb fois. En conclusion, cette réduction coûte $\mathcal{O}(npn_b)$.

Comme les pixels supprimés peuvent être très éloignés d'une ligne à l'autre, cela crée des distorsions importantes, comme montré ci-dessous où l'on supprime 240 pixels sur un total de 640 initialement par ligne. Pour remédier à ce problème, une idée consiste à supprimer la colonne de plus faible énergie c'est-à-dire celle dont la somme des énergies de ses pixels est minimale.

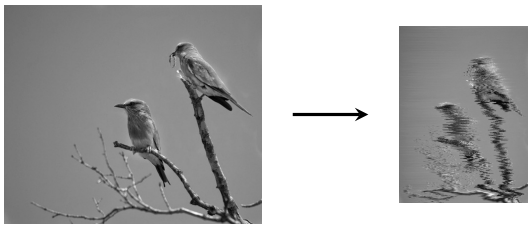

Figure 3 : Résultat de la méthode naïve

12 –Rédiger une fonction `énergie_colonnes(enrj:list) -> list` qui prend en argument la matrice énergie `enrj` associée à une image et qui renvoie une liste de taille p contenant l'énergie totale de chaque colonne.

```
def énergie_colonnes(enrj:list) -> list:
    n, p = dim(enrj)
    enrj_colonne = [0 for _ in range(p)]
    for i in range(n):
        for j in range(p):
            enrj_colonne[j] += enrj[i][j]
    return enrj_colonne
```

13 – En déduire une fonction `réduction_colonne(image:list, nb: int) -> list` qui renvoie une copie de la matrice `image` dans laquelle ont été supprimées les `nb` colonnes choisies selon la stratégie proposée.

```
def réduction_colonne(image:list, nb: int) -> list:
    enrj = énergie(image) # moins coûteux que dans for repetition
    enrj_col = énergie_colonnes(enrj)
    new = copie_im(image)
    n, p = dim(image)
    for répétition in range(nb):
        ind = indice_min(enrj_col)
```

```
        for i in range(n):
            retirer_indice(new[i], ind)
        retirer_indice(enrj_col, ind)
    return new
```

Le résultat obtenu est bien sûr meilleur (mais néanmoins perfectible). L'exemple ci-dessous supprime 300 pixels sur un total de 640 initialement par ligne.

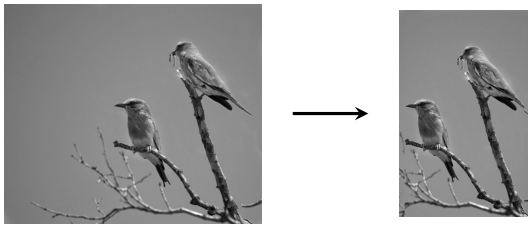

Figure 4 : Résultat de la méthode naïve

IV. Notion de chemin

On définit un chemin de pixels comme une suite de pixels en contact verticalement ou diagonalement (à gauche ou à droite) avec un premier pixel sur la première ligne de l'image et un dernier sur la dernière ligne de l'image. Un chemin possède exactement un pixel par ligne de l'image. L'énergie d'un chemin correspond à la somme des énergies des pixels de ce chemin.

Par exemple sur la matrice d'énergie E de 4×4 pixels représentée par le tableau ci-dessous, on exhibe un chemin quelconque, codé par une liste de n entiers représentant les colonnes des pixels du chemin.

2	1	4	8
2	5	6	7
1	7	9	8
3	1	1	4

`chemin = [1,2,1,1]` ⟶

2	**1**	4	8
2	5	**6**	7
1	**7**	9	8
3	**1**	1	4

L'objectif de la réduction est de déterminer un chemin d'énergie minimale pour ensuite supprimer les pixels correspondants de l'image. On envisage une approche gloutonne d'optimisation locale en déterminant le pixel d'énergie minimale de la première ligne (c'est le premier pixel du chemin) puis on prend comme second pixel celui d'énergie minimal parmi les trois (ou deux) pixels de la deuxième ligne juste en dessous du premier pixel (le plus à gauche en cas d'égalité) et ainsi de suite jusqu'à arriver à la dernière ligne de l'image.

14 – Donner le chemin obtenu en appliquant la stratégie gloutonne à l'image d'énergie E donnée sur l'exemple 4×4 ci-dessus.
Donner un exemple de matrice 3×3 composé uniquement de 0 et 1 pour laquelle cette stratégie n'est pas optimale.

Chemin glouton : `chemin = [1,0,0,1]`

Exemple non optimal :

0	1	1
1	1	0
1	1	0

---

15 – Proposer une fonction **récursive** `chemin_glouton(enrj: list, j: int) -> list` qui renvoie la liste des indices du chemin de plus faible énergie selon l'algorithme glouton.

```python
def chemin_glouton_rec(enrj, j):  # recursif
    if len(enrj) == 0:   # cas de base
        return []
    jmin, jmax = j-1, j+1
    if jmin < 0:              # gestion du bord gauche
        jmin = 0
    if jmax == len(enrj[0]):  # gestion du bord droit
        jmax = len(enrj[0])-1
    sous_tab = enrj[0][jmin:jmax+1]
    ind = indice_min(sous_tab) + jmin  # on re-décale l'indice
    return [ind] + chemin_glouton_rec(enrj[1:], ind)
```

16 – En déduire une fonction `réduction_glouton(image:list, nb: int) -> None` qui renvoie une copie de la matrice `image` en supprimant `nb` chemins choisis selon la stratégie gloutonne.

```python
def réduction_glouton(image: list, nb: int) -> list:
    new = copie_im(image)
    n, p = dim(image)
    for répétition in range(nb):
        enrj = np.array(énergie(new))
        chemin = chemin_glouton_rec(enrj, indice_min(enrj[0]))
        for i in range(n):
            indice = chemin[i]
            retirer_indice(new[i], indice)
    return new
```