

## TP 14 : APPLICATION DE DIJKSTRA AU TRAITEMENT D'IMAGE

## 1 Dijkstra

On rappelle que l'algorithme de Dijkstra travaille sur un graphe (orienté ou non) *pondéré* avec des poids positifs, représenté par listes d'adjacences, et calcule les distances et plus courts chemins de tous les sommets accessibles depuis un sommet source donné. Son pseudo-code est :

```
Dijkstra(G, s) :
    la distance de chaque sommet est infinie
    aucun sommet n'a de père
    les sommets sont non marqués
    s a pour distance 0
    tant qu'il reste un sommet non marqué :
        soit u le sommet non marqué de distance minimale
        on marque u
        pour chaque successeur v de u:
            on relâche l'arc (u,v), autrement dit :
            soit d' la distance de u + le poids w de l'arc (u,v)
            si d' est strictement inférieure à la distance de v:
                la distance de v devient d'
                le père de v devient u
    on renvoie les distances et les pères
```

Implémenter cet algorithme. On pourra le tester sur le graphe suivant :

```
G1 = [[(1,10), (4,5)],
      [(2,1), (4,2)],
      [(3,4)], [(2,6), (0,7)],
      [(1,3), (2,9), (3,2)]]
```

## 2 Sean carving

Le *seam carving* (ou recadrage intelligent), est un algorithme de redimensionnement d'image développé par Shai AVIDAN et Ariel SHAMIR. Cet algorithme redimensionne, non pas par une mise à l'échelle ou un recadrage classique, mais par une suppression des pixels dits "de moindre importance".

L'importance d'un pixel est en général mesurée par son contraste comparé à ses plus proches voisins, mais d'autres techniques, comme de détections de formes, peuvent-être utilisées.

On considère l'image suivante, dont on veut réduire la largeur de manière intelligente.



Pour cela on va supprimer successivement des « chemins » verticaux bien choisis dans l'image. Un chemin est une suite de coordonnées  $(0, y_0), (1, y_1), \dots, (n-1, y_{n-1})$  où  $n$  est la hauteur de l'image et pour tout  $k$ ,  $y_k \in \llbracket 0, p-1 \rrbracket$  où  $p$  est la largeur de l'image, avec la contrainte  $|y_{k+1} - y_k| \leq 1$ . On trace ci-dessous en rouge un chemin vertical :



Une image est représentée par un tableau numpy. Chaque pixel `img[i, j]` est une liste de longueur 3 codant la couleur (RGB). Afin de transformer une image en tableau numpy on utilise la commande suivante :

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

im_tour = np.array(plt.imread('BroadwayTower.png'))
```

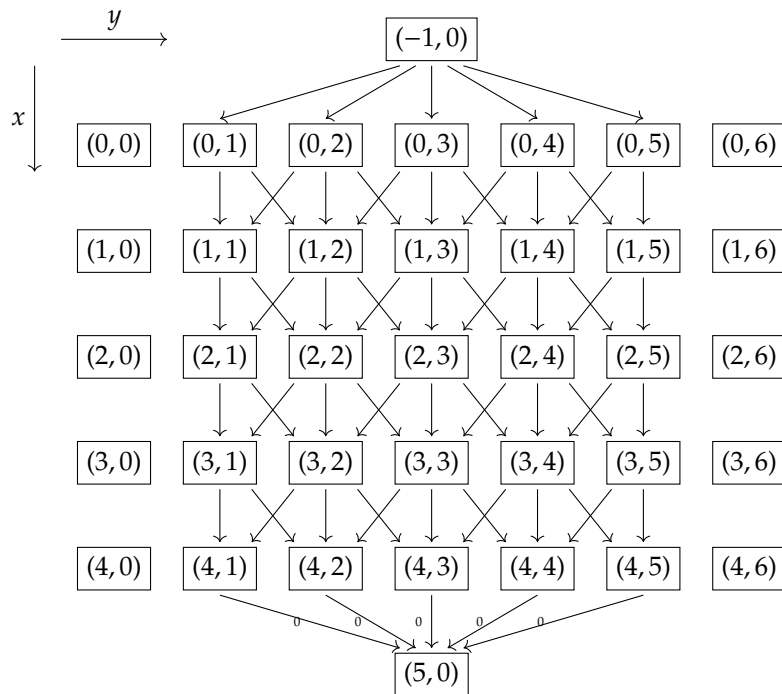
Pour récupérer la taille de l'image :

```
n, p, q = image.shape # n = nb de lignes, p = nb de colonnes, q = 3 en RGB
```

et pour afficher l'image :

```
plt.figure()
plt.imshow(im_tour)
plt.title('image de depart')
plt.show()
```

Nous allons voir l'image comme un graphe où chaque pixel est un sommet et de chaque sommet partent trois arcs, allant respectivement vers le pixel en bas à gauche, en bas et en bas à droite du sommet considéré. On évite d'ajouter des arcs vers les pixels du bord. Si on schématise une image comportant 5 lignes et 7 colonnes :



On a ajouté un sommet au-dessus et un sommet en-dessous de l'image, respectivement  $(-1, 0)$  et  $(n, 0)$  et des arcs reliant le sommet au-dessus à tous les pixels du bord supérieur de l'image, et reliant le sommet en-dessous à tous les pixels du bord inférieur de l'image. Cette astuce va nous permettre d'appliquer l'algorithme de Dijkstra en prenant pour sommet de départ  $(-1, 0)$  et pour sommet d'arrivée  $(n, 0)$ .

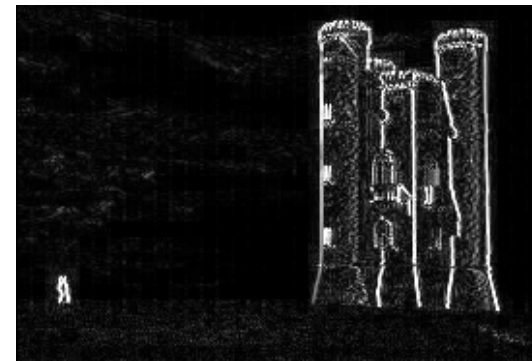
On décide d'attribuer à chaque pixel (excepté ceux des bords gauche et droit) une « énergie » correspondant à la norme de la différence du vecteur des couleurs des pixels situés sur la même ligne juste à droite et juste à gauche : l'élément  $(x, y)$  contient la norme de la différence entre le pixel  $(x, y + 1)$  et  $(x, y - 1)$ .

Rappel : si  $\vec{u} = (x, y, z)$  alors sa norme (pour le produit scalaire canonique) est  $N(u) = \sqrt{x^2 + y^2 + z^2}$ .

1. Écrire la fonction `deriv_x` qui prend en argument une image sous forme d'un tableau `numpy` et renvoie un tableau de même taille contenant l'énergie de chaque pixel (qu'on initialisera avec `np.zeros`). Pour les pixels des bords gauche et droit on affectera la valeur 0.

L'énergie du pixel est un bon candidat pour la longueur des arcs pointant vers ce pixel : en effet, un chemin empruntant un tel arc mettra en contact, une fois supprimé, les pixels  $(x, y - 1)$  et  $(x, y + 1)$ , il est donc logique que le poids de cet arc soit d'autant plus grand que cette différence est forte. On considérera que les arcs pointant vers le sommet tout en dessous ont un poids nul.

Pour notre image, si on trace l'énergie en noir et blanc :



On utilise la commande suivante pour un tracé en noir et blanc :

```
plt.imshow(energy, cmap=plt.cm.gray, vmin=0, vmax=1)
```

2. Écrire une fonction `arcs_sortants(energy, noeud)` qui pour un noeud  $(x, y)$  doit retourner la liste des arcs sortants du pixel  $(x, y)$  en accord avec la représentation de l'image sous forme de graphe expliquée précédemment. La fonction donnera aussi les arcs sortants pour les deux pixels (en haut et en bas) ajoutés artificiellement. La liste des arcs sortants sera donnée sous forme d'une liste de couples  $(\text{pixel}, \text{energie})$ , où `pixel` est le noeud vers lequel l'arc pointe et `energie` est l'énergie de ce pixel que l'on choisira comme poids de l'arc pour l'algorithme de Dijkstra.
3. Écrire une fonction `dijkstra_image` prenant en argument une image et renvoyant la distance minimale entre les noeuds "artificiels"  $(-1, 0)$  et  $(n, 0)$  avec  $n$  la hauteur de l'image, ainsi que l'ensemble des prédécesseurs. On utilisera des dictionnaires pour stocker les distances et les prédécesseurs. On stockera les sommets en attente dans un `set` `L`, contenant initialement le sommet source, puis à chaque étape on retirera le sommet  $u$  de `L` de distance minimale avec `L.remove(u)`, et pour chaque arc relâché  $(u, v)$  on ajoutera  $v$  avec `L.add(v)`.
4. Écrire une fonction `chemin` prenant en argument le dictionnaire des prédécesseurs, et les sommets de départ et d'arrivée et reconstruisant le chemin reliant les deux sommets.
5. Écrire une fonction `supprimer_chemin` prenant en argument une image et un chemin donné par la fonction précédente et renvoyant une nouvelle image construite en supprimant les pixels du chemin dans l'image de départ.
6. Réduire la largeur de l'image de 50 pixels en répétant 50 fois l'opération.