TP 6
Algorithme de Dijkstra
et application au traitement d'image

L'objet de ce TP est de comprendre et d'implémenter l'algorithme de Dijkstra (du mathématicien et informaticien néerlandais Edsger Dijkstra, 1930-2002, lauréat du prix Turing en 1972) permettant de déterminer le plus court chemin dans un graphe. On donnera ensuite une application de l'algorithme pour redimensionner une image de manière intelligente (« Seam carving », ou recadrage intelligent).

1 Introduction

Un graphe est un ensemble de points appelés *noeuds* et de segments appelés *arcs* reliant ces noeuds par paires. Si ces segments sont des flèches, on dit que le graphe est orienté. On se contentera ici de traiter le cas de graphes orientés. Ces arcs peuvent être pondérés pour représenter une distance entre noeuds, un coût, etc.

Un graphe peut servir à représenter un réseau informatique, un réseau de transports en commun, ou même une image, comme on le verra en application.

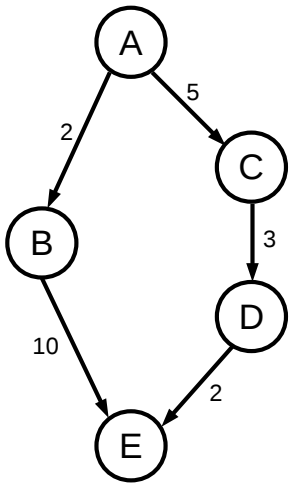


Figure 1 – Un graphe orienté pondéré

On définit un chemin dans un graphe orienté pondéré comme une suite de noeuds reliés deux à deux par des arcs, et la longueur d'un chemin comme la somme des longueurs des arcs de cette suite.

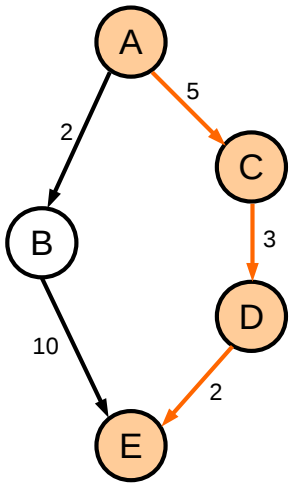


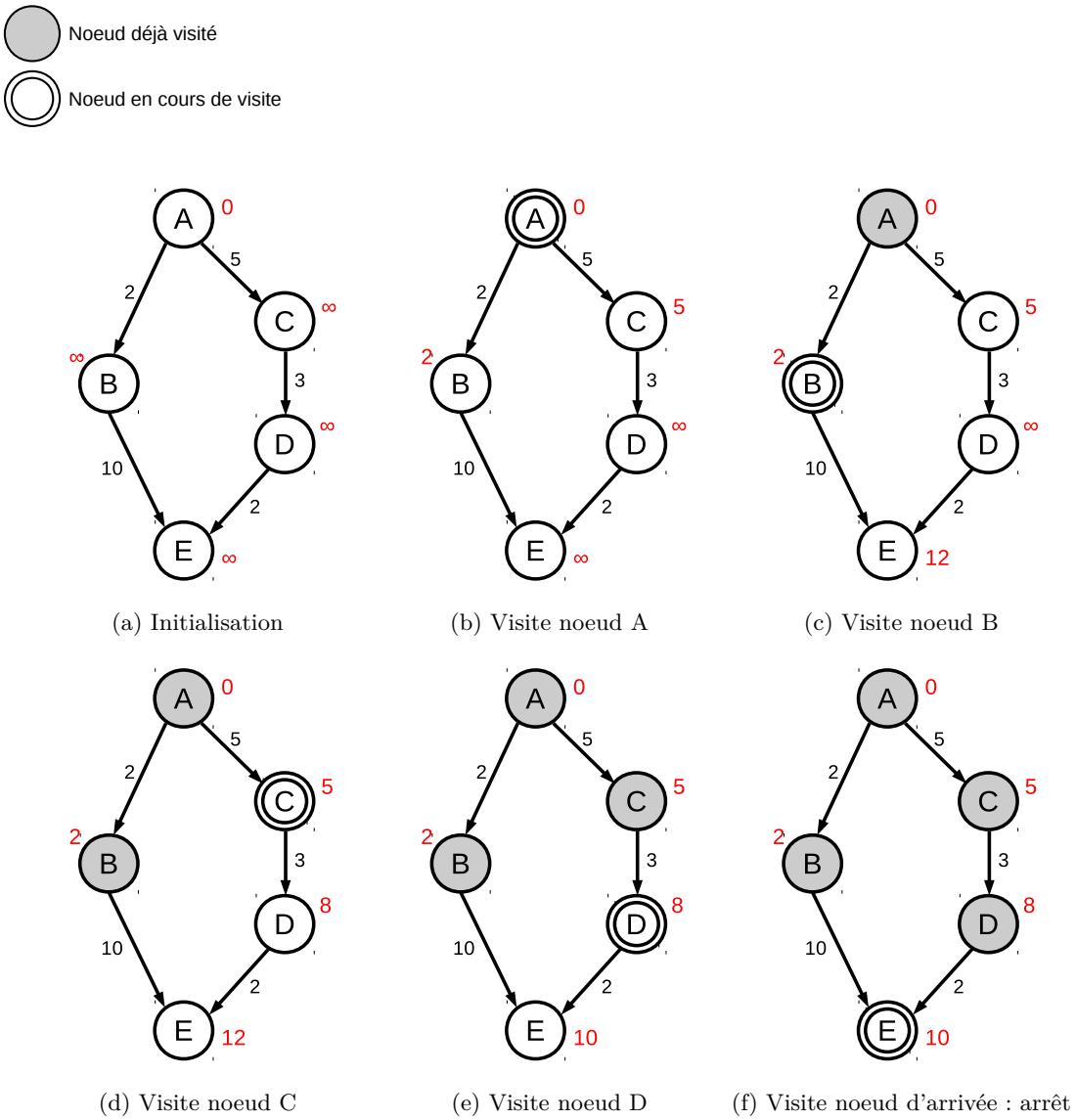
FIGURE 2 – Chemin A,C,D,E

L'algorithme de Dijkstra permet de résoudre le problème suivant : étant donné un graphe orienté pondéré, un noeud de départ et un noeud d'arrivée, trouver un chemin de longueur minimal allant du noeud départ au noeud d'arrivée.

2 Pseudo-algorithme

On donne ci-dessous l'algorithme de Dijkstra en *pseudo-code* permettant de déterminer la distance minimale d'un noeud de départ à un noeud d'arrivée.

```
1:  procedure DIJKSTRA(G,depart,arrivee)
2:      noeud_visites ← ∅
3:      pour chaque noeud n de G faire
4:          distance_min[n] ← +∞
5:      fin pour
6:      distance_min[depart] ← 0
7:      tant que noeuds_visites ne contient pas tous les noeuds de G faire
8:          noeud_courant ← noeud non visité de distance minimale
9:          noeud_visites ← noeud_visites ∪ {noeud_courant}
10:         si noeud_courant = arrivee alors
11:             quitter boucle
12:         fin si
13:         pour chaque arc sortant (successeur,longueur_arc) de noeud_courant faire
14:             distance ← distance_min[noeud_courant] + longueur_arc
15:             si distance < distance_min[successeur] alors
16:                 distance_min[successeur] ← distance
17:             fin si
18:         fin pour
19:     fin tant que
20:     retourner distance_min[arrivee]
21: fin procedure
```



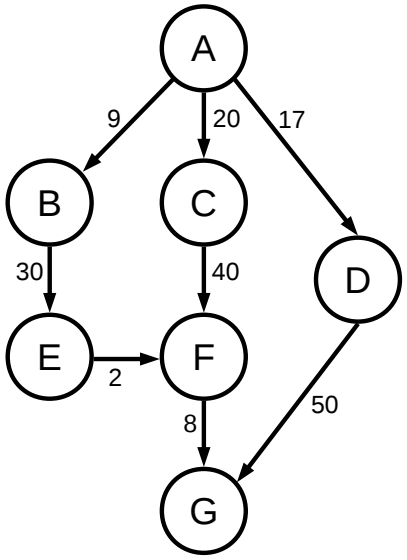
(a) Initialisation (b) Visite noeud A (c) Visite noeud B

(d) Visite noeud C (e) Visite noeud D (f) Visite noeud d'arrivée : arrêt

FIGURE 3 – Application de l'algorithme de Dijkstra

Si S et A dénotent respectivement le nombre de sommets et le nombre d'arcs du graphe, alors l'algorithme de Dijkstra a un coût en temps en $O((S + A) * ln(S))$, en utilisant des structures de données appropriées pour représenter le graphe et récupérer le noeud non visité de distance minimale.

Q1 Appliquer l'algorithme de Dijkstra « à la main » au graphe ci-dessous.



Q2 Proposer une modification de l'algorithme de Dijkstra pour que celui-ci retourne également un chemin de distance minimale reliant le noeud de départ au noeud d'arrivée.

3 Implémentation

3.1 Représentation d'un graphe

Il existe globalement deux grandes manières de représenter un graphe en machine.

3.1.1 Matrice d'adjacence

On peut considérer que le graphe comporte N noeuds numérotés de 0 à $N-1$, et représenter ses arcs par un tableau de nombres de dimension $N \times N$, de sorte que si l'élément $[i, j]$ du tableau vaut -1, il n'existe pas d'arc allant du noeud i au noeud j, et s'il est positif, il existe un tel arc et sa longueur est donnée par ce nombre.

Q3 Donner la représentation sous forme de matrice d'adjacence du graphe donné en question 1, et écrire la commande Python pour la construire.

3.1.2 Liste d'adjacence

On peut représenter le graphe en associant à chaque noeud n une liste d'arcs, un arc étant un couple (s, l) où s est un noeud et l la longueur de l'arc allant du noeud n au noeud s. Considérons que les noeuds sont numérotés, si `G[5]` vaut `[(0,20),(3,4)]`, cela signifie que du noeud 5 partent deux arcs, un allant vers le noeud 0, de longueur 20, et un autre allant vers le noeud 3, de longueur 4.

Q4 Donner la représentation sous forme de liste d'adjacence du graphe donné en question 1, et écrire la commande Python pour la construire.

3.2 Structures de données utiles

On va utiliser deux structures de données dont l'implémentation est fournie, d'une part pour associer à chaque noeud une distance, et d'autre part pour récupérer efficacement le noeud non encore visité de distance minimale.

3.2.1 Dictionnaire

Un *dictionnaire* ou *tableau associatif* est une structure de données permettant d'associer à un ensemble de *clés* (qui peuvent être des nombres, chaînes de caractères ou tuples) une valeur (qui peut être un nombre, une liste, un objet, etc.). Les dictionnaires sont implémentés nativement en Python, et peuvent se manipuler de la manière suivante.

Listing 1 – Exemple d'utilisation d'un dictionnaire

```
1  # Creation d'un dictionnaire associant l'entier 10 a la cle
2  # 'cle1', et la liste [1,2,3] au couple (110,32)
3  dico = {'cle1': 10, (110,32):[1,2,3]}
4
5  # Ajout de l'association 'cle2' : 2
6  dico['cle2'] = [2]
7
8  # Affichage de la valeur associe a la cle (110,32)
9  print(dico.get((110,32)))
10
11 # Si aucune valeur n'est associee a une cle, la fonction get
12 # retourne None
13 valeur = dico.get('cle3')
14 if valeur is None:
15     print("Aucune␣valeur␣associee␣a␣\'cle3\'␣!")
```

Dans le cas de l'algorithme de Dijkstra, on utilisera un dictionnaire pour la variable *distance_min* du pseudo-code. Plutôt que d'associer une valeur arbitrairement grande à tous les noeuds du graphe en guise d'initialisation, on considèrera simplement que si aucune valeur n'a encore été associée à un noeud il est à distance $+\infty$ du noeud de départ.

3.2.2 File de priorité

Une file de priorité est une structure de données abstraite sur laquelle peuvent s'effectuer trois opérations :

1. insérer un élément avec une priorité donnée
2. récupérer et retirer de la file l'élément ayant la priorité la plus faible
3. tester si la file est vide

Dans le cas de l'algorithme de Dijkstra, on aura besoin d'une quatrième opération :

4. mettre à jour la priorité d'un élément déjà inséré dans la file

On dit que cette structure de données est abstraite car on se fiche de la manière dont elle est implémentée (à partir d'un tableau de taille variable, ou d'une liste, etc.), seules

nous intéressent les opérations qu'on peut effectuer sur celle-ci. On aurait pu implémenter une telle structure de données à partir d'une liste, qu'on maintiendrait triée à chaque insertion. En pratique, elle est souvent implémentée autrement, de sorte à garantir que le coût amorti de chaque opération soit à peu près constant, indépendamment de la taille de la file.

On fournit ici une implémentation d'une file de priorité à partir du module Python `heapq`, qui peut se manipuler de la manière suivante :

Listing 2 – Exemple d'utilisation de la file de priorité

```
# Creation d'une file vide
file_priorite = PriorityQueue()

# Insertion de quelques elements dans la file
file_priorite.insert(43, 10)
file_priorite.insert((2,54), 4.2)
file_priorite.insert('element', 8)

# insert permet egalement de mettre a jour la priorite d'un
# element de la file, s'il est deja present dans celle-ci.
file_priorite.insert((2,54), 9)

# On va recuperer les elements de la file un a un et les
# afficher, jusqu'a ce qu'elle soit vide.
# On ne peut pas tester directement si la file est vide, il faut
# essayer de recuperer un element : si c'est None, c'est qu'elle
# est vide.
while True:
    element = file_priorite.pop()
    if element is None:
        # La file est vide: on quitte la boucle
        break
    print(element)
```

Remarque : l'implémentation proposée de la file de priorité ne permet pas d'insérer des éléments tels que des listes. Sont autorisés néanmoins des chaînes de caractères, des nombres, et des tuples.

3.2.3 Implémentation

On considère dorénavant que le graphe est donné sous la forme d'une liste d'adjacence.

Q5 Écrire tout simplement une fonction `arcs_sortants(graphe, noeud)` retournant la liste des arcs sortant d'un noeud pour le graphe donné.

Q6 On donne ci-dessous en pseudo-code l'algorithme de Dijkstra adapté aux structures de données présentées précédemment. Appliquer « à la main » l'algorithme ci-dessous au

graphe de la question 1, en donnant à chaque itération les valeurs de `noeud_visites`, `distance_min` et `file_priorite`.

```
1: procedure DIJKSTRA(G,depart,arrivee)
2:     noeud_visites ← ∅
3:     file_priorite ← file vide
4:     ajouter depart à la file de priorité, avec la priorité 0
5:     distance_min ← dictionnaire vide
6:     distance_min[depart] ← 0
7:     tant que file_priorite n'est pas vide faire
8:         (distance, noeud_courant) ← récupérer et enlever min de la file de priorité
9:         noeud_visites ← noeud_visites ∪ {noeud_courant}
10:        si noeud_courant = arrivee alors
11:            quitter boucle
12:        fin si
13:        si noeud_courant n'a pas encore été visité alors
14:            pour chaque arc sortant (successeur,longueur_arc) de noeud_courant faire
15:                distance ← distance_min[noeud_courant] + longueur_arc
16:                si distance_min ne contient pas successeur ou distance < distance_min[successeur] alors
17:                    distance_min[successeur] ← distance
18:                    mettre à jour ou insérer (successeur, distance) dans la file de priorité
19:                fin si
20:            fin pour
21:        fin si
22:    fin tant que
23:    retourner distance_min[arrivee]
24: fin procedure
```

Q7 Écrire une fonction `def dijkstra(G, depart, arrivee)` retournant la distance minimale du noeud `depart` au noeud `arrivee`. On supposera définie et on utilisera la fonction `arcs_sortants` pour récupérer les arcs sortant d'un noeud. Vérifier sur le graphe de la question 1 que votre programme fonctionne correctement.

Remarque : le fait de n'utiliser que la fonction `arcs_sortants` *permet de s'affranchir de la manière dont est stocké le graphe dans l'algorithme de Dijkstra, et d'écrire un code général qui fonctionnera si le graphe est stocké différemment : il suffira de réimplementer la fonction* `arcs_sortants`*.*

Q8 Modifier la fonction précédente pour qu'elle retourne également le plus court chemin du noeud de départ au noeud d'arrivée. La fonction doit ainsi renvoyer un couple `(chemin,distance)`.

4 Application au redimensionnement d'images

4.1 Chemins verticaux

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

FIGURE 4 – Image test (auteur : Newton2 de Wikipedia anglais)

Pour cela, on va supprimer successivement des « chemins » verticaux bien choisis dans l'image. Un chemin est une suite de coordonnées $(x_0, 0)(x_1, 1)...(x_{h-1}, h-1)$ où h est la hauteur de l'image, et pour tout n, x_n est un entier positif strictement inférieur à la largeur, et $|x_{n+1} - x_n| \leq 1$.

FIGURE 5 – Un chemin vertical

4.2 Une image comme un graphe

On peut voir une image comme un graphe (Figure 6). Chaque pixel est un noeud, et de chaque noeud partent trois arcs, allant respectivement vers le pixel en bas à gauche, en

bas, et en bas à droite du noeud considéré. On évite d'ajouter des arcs vers les pixels du bord, pour des raisons qu'on expliquera par la suite.

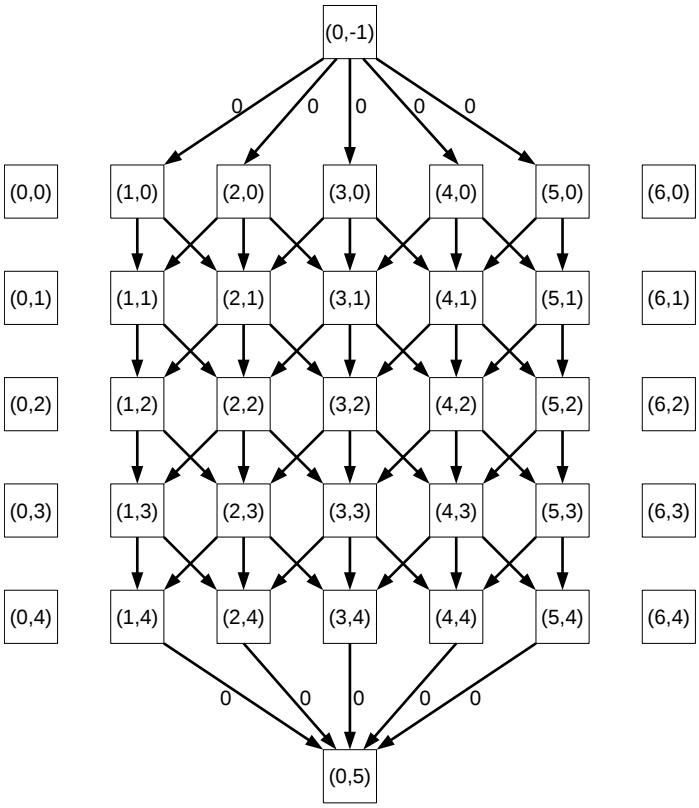


FIGURE 6 – Graphe associé à une image 7×5

On a ajouté artificiellement un noeud au-dessus et en-dessous de l'image, respectivement $(0, -1)$ et $(0, h)$ et des arcs de longueur nulle reliant le noeud au-dessus aux pixels du bord supérieur de l'image, et le noeud en-dessous aux pixels du bord inférieur de l'image. Cette astuce va nous permettre d'appliquer directement l'algorithme de Dijkstra, qui demande un noeud de départ et un noeud d'arrivée, alors qu'on cherche un plus court chemin reliant n'importe quel pixel du bord supérieur à n'importe quel pixel du bord inférieur.

À présent, quelle longueur donner aux arcs sortant de chaque pixel, de sorte que le plus court chemin perturbe peu l'image ?

4.3 Énergie d'une image

On donne une fonction `deriv_x(image)` définie comme ci-dessous :

Listing 3 – Énergie

```
1 def deriv_x(image):
2     res = np.zeros((image.shape[0],image.shape[1]))
3     for y in range(image.shape[0]):
4         for x in range(1,image.shape[1]-1):
5             res[y,x] = lg.norm(image[y,x+1,:]-image[y,x-1,:])
```

Q9 Exécuter le code ci-dessous.

```
1 image = mpimg.imread('BroadwayTower.png')
2 energy = deriv_x(image)
3 energyimg = np.array([energy,energy,energy])
4 energyimg = np.swapaxes(energyimg,0,1)
5 energyimg = np.swapaxes(energyimg,1,2)
6 plt.imshow(energyimg)
```

Qu'effectue la fonction `deriv_x`, et que représente le tableau qu'elle retourne ? Proposer alors une valeur pour la longueur d'un arc allant vers un noeud (x, y).

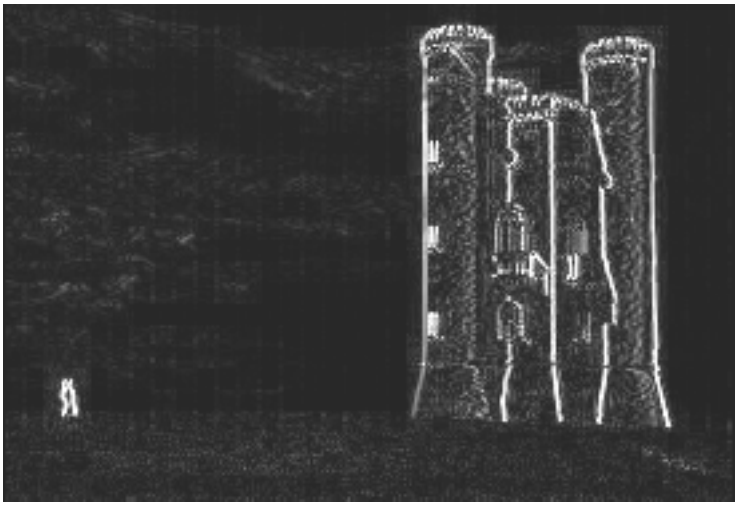


FIGURE 7 – « Énergie » de l'image

Q10 Réimplémenter la fonction `arcs_sortants(graphe, noeud)`, en considérant que le graphe est donné par le couple `(image,energy)`, et le noeud est un couple `(x,y)` représentant un pixel de l'image. La fonction 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 (d'un pixel sortent 3 arcs, sauf près des bords où il n'en sort que deux, et on ajoute artificiellement un noeud au-dessus et un noeud en-dessous de l'image).

Q11 Appliquer l'algorithme de Dijkstra pour récupérer un plus court chemin dans l'image, puis utiliser la fonction `tracer_chemin` pour l'afficher.

Q12 À l'aide de la fonction `supprimer_chemin`, réduire la largeur de l'image de 100 pixels en répétant successivement toute l'opération et en affichant à chaque itération l'image courante et le chemin qui va être supprimé.

(a) Image originale

(b) Redimensionnement classique

(c) Redimensionnement par seam carving