

# Graphes, coloration et plus courts chemins

INFORMATIQUE COMMUNE - TP n° 2.4 - Olivier Reynet

## À la fin de ce chapitre, je sais :

- ✎ parcourir un graphe en largeur pour le colorer
- ✎ coder l'algorithme de Dijkstra
- ✎ utiliser une file de priorités

## A Coloration en largeur

On se donne un graphe  $g$  sous la forme d'une liste d'adjacence ainsi qu'une liste de couleur utilisables pour colorer une graphe avec les bibliothèques `matplotlib` et `networkx`.

```
g = [[1, 3, 14], [0, 2, 4, 5], [1, 5, 7], [0, 4, 8, 13, 14], [1, 3, 5, 8],  
     [1, 2, 4, 6, 7], [5, 7], [2, 5, 6], [3, 4, 9, 13], [8, 10, 13], [9, 11, 13],  
     [10, 12, 13], [11, 13], [3, 8, 9, 10, 11, 12, 14], [0, 3, 13]]  
colors = ["deeppink", "darkturquoise", "yellow", "dodgerblue", "magenta", "  
          darkorange", "lime"]
```

L'objectif de cette section est de colorer le graphe à l'aide d'un algorithme glouton et d'un parcours en largeur. C'est un algorithme glouton dont l'ordre est l'ordre du parcours en largeur. Pour chaque sommet découvert, on choisit la première couleur disponible dans la liste.

- A1. Écrire une fonction de prototype `couleur_suivante(utilisees, couleurs)` dont les paramètres sont la liste des couleurs utilisées par les voisins d'un sommet et la liste de toutes les couleurs. Cette fonction renvoie la première couleur disponible parmi celles qui ne sont pas utilisées par un voisin. S'il n'y en a aucune, elle renvoie `None`.

### Solution :

```
def couleur_suivante(utilisees, couleurs):  
    for c in couleurs:  
        if c not in utilisees:  
            return c  
    return None
```

- A2. Écrire une fonction de prototype `bfs_color(g, start, couleurs)` qui parcourt en **largeur** un graphe  $g$  sous la forme d'une liste d'adjacence à partir du sommet `start`. Cette fonction renvoie une coloration du graphe sous la forme d'un dictionnaire dont les clefs sont les numéros du sommet et les valeurs les couleurs de ces sommets. Par exemple :

```
color_map = {5: 'deeppink', 1: 'darkturquoise', 2: 'yellow', 4: 'yellow'...}
```

**Solution :**

```
def bfs_color(g, start, couleurs):
    color_map = dict()
    decouverts = [False for _ in range(len(g))]
    decouverts[start] = True
    F = [start]
    while len(F)>0:
        s = F.pop(0)
        v_colors = [] # couleurs des voisins
        for v in g[s]:
            if v in color_map:
                v_colors.append(color_map[v]) # couleur déjà utilisée par
                les voisins
            if not decouverts[v]:
                F.append(v)
                decouverts[v] = True
        color_map[s] = couleur_suivante(v_colors, couleurs) # choisir une
        couleur pour s
    return color_map
```

A3. Quelle est la complexité de cet algorithme de coloration dans le pire des cas?

**Solution :** Dans le pire des cas, le graphe est complet. Si l'algorithme utilisait une file du module Queue, on obtiendrait  $O(n(n-1+c)) = O(n^2)$  si le nombre de couleurs dans la liste des couleurs est  $c$  et qu'on peut le négliger.

Attention : en Python, l'opération `F.pop(0)` est en  $O(n)$ . C'est un coût «caché» si on n'y prête pas attention qui amène à  $O(n^2 + m)$  si  $m$  est le nombre d'arêtes.

Les fonction ci-dessous permettent :

1. de transformer un graphe sous la forme d'une liste d'adjacence en un objet graphe de la bibliothèque networkx,
2. de tracer le graphe coloré, le dictionnaire `color_map` est le résultat de l'algorithme glouton de coloration.

```
import networkx as nx
from matplotlib import pyplot as plt

def ladj_to_nx(g):
    gx = nx.Graph()
    n = len(g)
    gx.add_nodes_from([i for i in range(n)])
    for i in range(n):
        for v in g[i]:
            gx.add_edge(i, v)
    return gx

def show(g, color_map):
    color_nodes = [""] for _ in range(len(g))
    for k, v in color_map.items():
        color_nodes[k] = v
```

```

gx = ladj_to_nx(g)
nx.draw_networkx(gx, node_color=color_nodes, with_labels=True)
plt.show()

```

A4. En utilisant les fonctions ci-dessus, tracer le graphe coloré par l'algorithme glouton.

**Solution :**

```

colors = ["deeppink", "darkturquoise", "yellow", "dodgerblue", "magenta", "darkorange", "lime"]
g = [[1, 3, 14], [0, 2, 4, 5], [1, 5, 7], [0, 4, 8, 13, 14], [1, 3, 5, 8],
     [1, 2, 4, 6, 7], [5, 7], [2, 5, 6], [3, 4, 9, 13], [8, 10, 13], [9, 11, 13],
     [10, 12, 13], [11, 13], [3, 8, 9, 10, 11, 12, 14], [0, 3, 13]]
color_map= bfs_color(g, 5, colors)
print(color_map)
show(g,color_map)

```

## B Plus courts chemins : algorithme de Dijkstra

On considère maintenant un graphe non orienté pondéré  $g$  représenté par sa liste d'adjacence : les sommets sont des villes et les poids représentent les distances en kilomètres entre les villes. On cherche à calculer les distances les plus courtes depuis la ville d'indice 0 vers toutes les directions possibles.

L'algorithme de Dijkstra nécessite une file de priorités. L'implémentation via une liste Python n'est pas optimale en termes de complexité (cf. cours). C'est pourquoi, on utilise la bibliothèque `queue` qui contient notamment une file de priorités présentant une complexité optimale. Voici un exemple d'utilisation :

```

import queue
pq = queue.PriorityQueue()
pq.put((d, s)) # enfiler le sommet s à la distance de d
delta, s = pq.get() # défiler le sommet s le plus proche à la distance d

```

Pour visualiser, on utilisera la bibliothèque `networkx` comme suit :

```

import matplotlib.pyplot as plt
import networkx as nx
import math

import numpy as np
from matplotlib import cm

def ladj_to_nx(g):
    gx = nx.Graph()
    n = len(g)
    gx.add_nodes_from([i for i in range(n)])
    for i in range(n):
        for v, d in g[i]:
            gx.add_edge(i, v, weight=d)
    return gx

```

```
def show(g):
    n = len(g)
    color_list = list(iter(cm.rainbow(np.linspace(0, 1, n))))
    gx = ladj_to_nx(g)
    pos = nx.spring_layout(gx)
    nx.draw_networkx(gx, pos, node_color=color_list, with_labels=True)
    edge_labels = nx.get_edge_attributes(gx, "weight")
    nx.draw_networkx_edge_labels(gx, pos, edge_labels)
    plt.show()

g = [[...], [...], [...], ... , [...]]
show(g)
```

### B1. Pour le graphe

```
gp = [(1, 2), (2, 5), (4, 7), (5, 4)], [(0, 2), (2, 2)],
      [(0, 5), (1, 2), (3, 1)], [(2, 1), (4, 14), (5, 1)],
      [(0, 7), (3, 14)], [(0, 4), (3, 1)]]
```

effectuer à la main l'algorithme de Dijkstra à partir du sommet 4. On représentera la solution sous la forme d'un tableau dont les colonnes sont les sommets du graphe.

$\Delta$		4	0	3	1	2	5
<b>Solution :</b>	$\{\}$	0	7	14	$+\infty$	$+\infty$	$+\infty$
	$\{0\}$	.	7	14	9	12	11
	$\{0, 1\}$	.	7	14	9	11	11
	$\{0, 1, 2\}$	.	7	12	9	11	11
	$\{0, 1, 2, 5\}$	.	7	12	9	11	11
	$\{0, 1, 2, 5, 3\}$	.	7	12	9	11	11

### B2. Écrire une fonction de prototype `pq_dijkstra(g, start)` qui implémente l'algorithme de Dijkstra (cf. algorithme 1). Cette fonction renvoie le tableau des distances les plus courtes depuis le sommet `start` ainsi que le tableau des parent de chaque sommet sur le chemin.

#### Solution :

```
def pq_dijkstra(g, start):
    n = len(g)
    pq = queue.PriorityQueue()
    d = [math.inf for _ in range(n)]
    d[start] = 0
    parents = [i for i in range(n)]
    pq.put((d[start], start))
    for _ in range(n):
        delta, s = pq.get() # transfert
        for v, dv in g[s]:
            if d[v] > delta + dv: # mise à jour de la distance
                d[v] = delta + dv
                pq.put((d[v], v))
                parents[v] = s
    return d, parents
```

- B3. Quelle est la complexité de la fonction `dijkstra`? On fait l'hypothèse que les opérations défiler et enfiler sont de complexité  $O(\log n)$ .

**Solution :** La complexité vaut alors  $O((n + m) \log n)$ , car le transfert de chaque sommet et l'insertion dans la file de priorités se font en  $\log n$ .

- B4. Écrire une fonction de prototype `build_path(parents, end)` qui renvoie le chemin le plus court du sommet de départ au sommet `end`. Cette fonction utilise le dictionnaire `parents` qui a été créé lors de l'exécution de l'algorithme de Dijkstra.

**Solution :**

```
def build_path(parents, end):
    path = []
    while parents[end] != end:
        path.append(end)
        end = parents[end]
    path.append(end)
    path.reverse()
    return path
```

- B5. Écrire une fonction de prototype `build_d_graph(g, parents)` qui construit le graphe des plus courts chemins, c'est-à-dire le graphe construit à partir de `g` où l'on ne conserve que les arêtes sélectionnées par l'algorithme de Dijkstra. Le résultat peut être visualisé sur la figure 1.

**Solution :**

```
def build_d_graph(g, parents):
    n = len(parents)
    dg = [[] for _ in range(n)]
    for s in range(n):
        for v, d in g[s]:
            p = parents[v]
            if p == s:
                dg[s].append((v, d))
                dg[v].append((s, d))
    return dg
```

- B6. Tester l'algorithme sur le graphe

```
g = [[(1, 393), (3, 240), (14, 209)], [(0, 393), (2, 290), (4, 146), (5, 221)], [(1, 290), (5, 244), (7, 195)], [(0, 240), (4, 105), (8, 216), (14, 102), (13, 254)], [(3, 105), (5, 258), (1, 146), (8, 217)], [(4, 258), (6, 184), (2, 244), (7, 216), (1, 221)], [(5, 184), (7, 114)], [(2, 195), (6, 114), (5, 216)], [(3, 216), (9, 113), (4, 217), (13, 113)], [(8, 113), (10, 60), (13, 116)], [(11, 69), (9, 60), (13, 155)], [(10, 69), (12, 71), (13, 216)], [(11, 71), (13, 244)], [(12, 244), (11, 216), (3, 254), (8, 113), (9, 116), (10, 155), (14, 147)], [(0, 209), (3, 102), (13, 147)]]
```

**Algorithme 1** Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

---

```

1: Fonction DIJKSTRA( $G = (S, A, w), s_0$ )           ▷ Trouver les plus courts chemins à partir de  $a \in S$ 
2:    $n \leftarrow |S|$                                    ▷ l'ordre du graphe
3:    $\Delta \leftarrow s_0$                                ▷  $\Delta$  est l'ensemble des sommets dont on connaît la distance à  $s_0$ 
4:    $\Pi \leftarrow$  tableau vide (taille  $n$ )             ▷  $\Pi[s]$  est le parent de  $s$  dans le plus court chemin de  $s_0$  à  $s$ 
5:    $d \leftarrow$  tableau vide (taille  $n$ )               ▷ l'ensemble des distances au sommet  $s_0$ 
6:    $\forall s \in S, d[s] \leftarrow w(s_0, s)$              ▷  $w(s_0, s) = +\infty$  si  $s$  n'est pas voisin de  $s_0$ , 0 si  $s = s_0$ 
7:   tant que  $\bar{\Delta}$  n'est pas vide répéter             ▷  $\bar{\Delta}$  : sommets dont la distance n'est pas connue
8:     Choisir  $u$  dans  $\bar{\Delta}$  tel que  $d[u] = \min(d[v], v \in \bar{\Delta})$    ▷ Choix glouton!
9:      $\Delta = \Delta \cup \{u\}$                            ▷ On prend la plus courte distance à  $s_0$  dans  $\bar{\Delta}$ 
10:    pour chaque voisin  $x$  de  $u$  répéter
11:      si  $d[x] > d[u] + w(u, x)$  alors
12:         $d[x] \leftarrow d[u] + w(u, x)$                ▷ Mises à jour des distances des voisins
13:         $\Pi[x] \leftarrow u$                            ▷ Pour garder la tracer du chemin le plus court
14:  renvoyer  $d, \Pi$ 

```

---

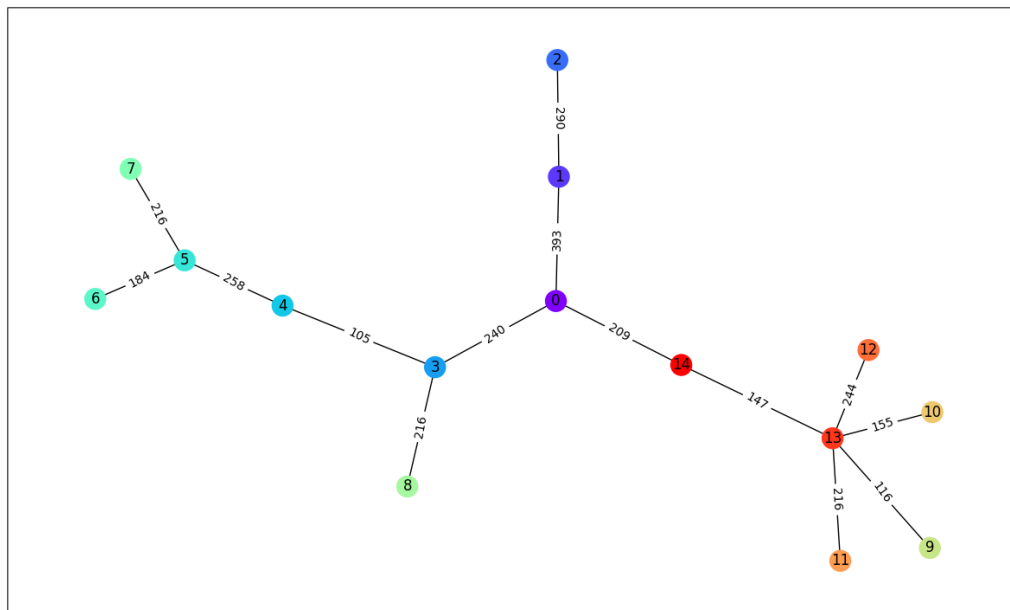


FIGURE 1 – Graphe issu de l'algorithme de Dijkstra : seules les arêtes sélectionnées par l'algorithme ont été conservées