Graphes, coloration et plus courts chemins

INFORMATIQUE COMMUNE - TP n° 2.6 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ colorer un graphe d'une manière valide
- ☞ coder l'algorithme de Dijkstra à l'aide d'une file de priorités
- ☞ utiliser une structure parents pour remonter le plus court chemin dans un graphe

A Coloration gloutonne

Ⓡ On rappelle qu'une coloration valide d'un graphe est une coloration où les sommets voisins n'ont pas la même couleur. Le nombre chromatique du graphe noté χ est le nombre minimal de couleurs nécessaires pour colorer de manière valide le graphe.

La coloration de graphe est un problème qui se rencontre dans de nombreux domaines. On peut citer notamment : l'attribution de fréquences dans les télécommunications, la conception de puces électroniques ou l'allocation de registres en compilation.

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'algorithme de Welsh-Powell est un algorithme de coloration de graphe glouton. Il utilise le degré des sommets pour définir l'ordre avec lequel il colore le graphe. Les sommets dont le degré est le plus élevé sont colorés en premier en choisissant toujours la première couleur disponible dans la liste des couleurs disponibles.

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`. On n'utilisera **pas** l'opérateur `in` pour tester l'appartenance d'un élément à une liste.

Solution :

```
def couleur_suivante(utilisees, couleurs):
    for c in couleurs:
```

```
            i = 0
            while i < len(utilisees) and utilisees[i] != c:
                i += 1
            if i == len(utilisees):
                return c
        return None


    def couleur_suivante(utilisees, couleurs):
        for c in couleurs:
            c_in = False
            for u in utilisees:
                if u == c:
                    c_in = True
            if not c_in:
                return c
        return None
```

A2. Écrire une fonction de prorotype `WP_color(g, couleurs)` qui implémente l'algorithme de Welsh-Powell : l'ordre glouton des sommets est celui des degrés décroissants. Le graphe `g` est une liste d'adjacence. Cette fonction renvoie une coloration du graphe sous la forme d'une liste dont les indices sont les sommets du graphe et les éléments les couleurs de ces sommets. Par exemple : `color_map = ['deeppink', 'darkturquoise', 'yellow', 'yellow']`.

Solution :

```
    def WP_color(g, couleurs):
        sommets = [(len(g[i]), i) for i in range(len(g))]
        sorted(sommets, reverse=True)
        color_map = ["" for _ in range(len(g))]
        i = 0
        while i < len(sommets):
            degre, s = sommets[i]
            v_colors = [] # couleurs des voisins
            for v in g[s]:
                if color_map[v] != "":
                    v_colors.append(color_map[v]) # couleur déjà utilisée par
                        les voisins
            color_map[s] = couleur_suivante(v_colors, couleurs) # choisir une
                couleur pour s
            i += 1
        return color_map
```

A3. En supposant que la complexité de la fonction couleur suivante est linéaire en fonction de l'ordre du graphe [1], quelle est la complexité de cet algorithme de coloration dans le pire des cas ?

1. c'est-à-dire on néglige le nombre de couleur devant le nombre de sommets du graphe

Solution : Dans le pire des cas, le graphe est complet. On parcours tous les sommets et tous les voisins des sommets. On a donc : $C(n) = n\log n + \sum_{i=0}^{n-1}\left(n + \sum_{v\in g[i]} c\right) = n\log n + n^2 + n + \sum_{i=0}^{n-1} \deg(i) = O(n^2+m) = O(n^2)$.

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, c in enumerate(color_map):
        color_nodes[k] = c
    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. On pourra également tester sur le graphe de Petersen. Combien de couleurs sont-elles nécessaires ?

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= WP_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 Dijsktra 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 delta
```

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.

Solution :

Δ	4	0	3	1	2	5
{}	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 Dijsktra (cf. algorithme **??**). Cette fonction renvoie le tableau des distances les plus courtes depuis le sommet `start` ainsi que le tableau des parents de chaque sommet sur les chemins.

Solution :

```
def pq_dijkstra(g, start):
    n = len(g)
    pq = queue.PriorityQueue()
    d = [math.inf for _ in range(n)] # l'infini ne peut pas être un minimum
        !
    d[start] = 0
    parents = [i for i in range(n)]
    pq.put((d[start], start))
    for _ in range(n):
        delta, s = pq.get()  # phase de transfert  O(log n)
        for v, dv in g[s]: # phase de mise à jour de la distance
            if d[v] > delta + dv:
                d[v] = delta + dv
                pq.put((d[v], v))   # O(log n)
                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 $\log n$.

$$C(n) = \sum_{i=0}^{n-1}\left(\log n + \sum_{v\in g[i]} \log n\right) = n\log n + \log n \sum_{i=0}^{n-1} \deg(i) = n\log n + m\log n = O((n+m)\log n).$$

B4. Écrire une fonction de prototype `parents_vers_chemin(parents, s)` qui renvoie le chemin le plus court du sommet de départ au sommet `s`. Cette fonction utilise le tableau `parents` qui a été créé lors de l'exécution de l'algorithme de Dijsktra.

Solution :

```
def parents_vers_chemin(parents, s):
```

```
    chemin = []
    while parents[s] != s:
        chemin.append(s)
        s = parents[s]
    R = []
    for k in range(len(chemin)): # retourner la liste
        R.append(chemin[n-1-k])
    return R
```

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 peu être visualisé sur la figure 1.

Solution :

```
def pcc_graphe(g, parents):
    n = len(parents)
    dg = [[] for _ in range(n)]
    for s in range(n):
        for v, d in g[s]:
            if s == parents[v]: # s fait parti du chemin
                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)]]
```

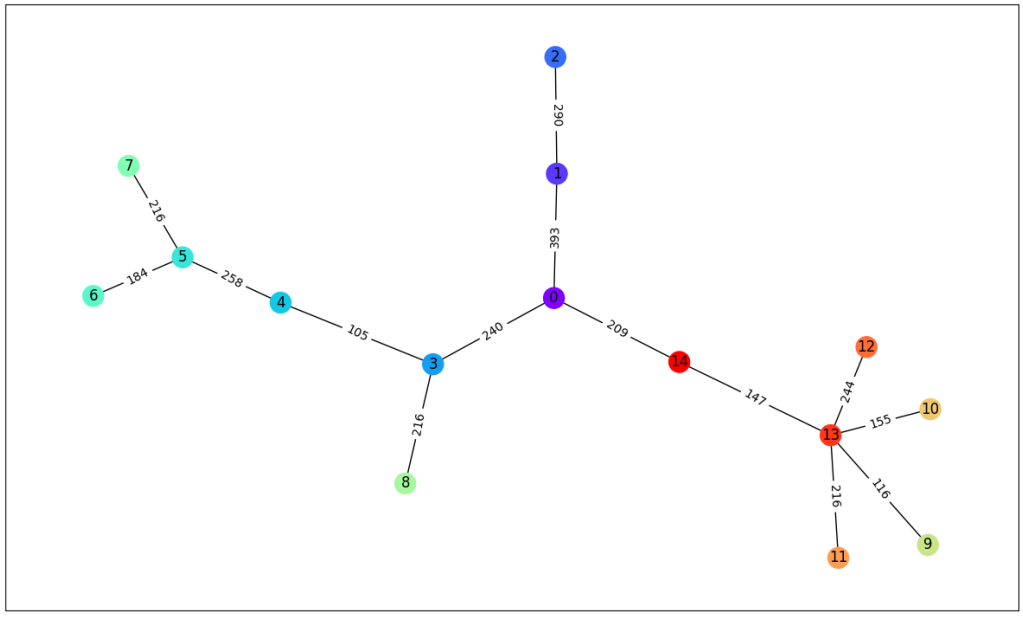

FIGURE 1 – Graphe issu de l'algorithme de Dijkstra : seules les arêtes sélectionnées par l'algorithme ont été conservées