# Graphes, coloration et plus courts chemins

INFORMATIQUE COMMUNE - TP nº 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 gloutonne

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. La stratégie gloutonne est la suivante : pour chaque sommet découvert, on choisit la première couleur disponible dans la liste.

A1. Écrire une fonction de prototype `next_possible_color(used_colors, colors)` 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 :**

```python
def next_possible_color(used_colors, colors):
    for c in colors:
        if c not in used_colors:
            return c
    return None
```

---

A2. Écrire une fonction de prorotype `greedy_color(g, start, colors)` qui parcours 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 :**

```python
def greedy_color(g, start, colors):
    color_map = {}
    E = []
    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 v not in E:
                F.append(v)
                E.append(v)
        color_map[s] = next_possible_color(v_colors, colors) # choisir une
            couleur pour s
    return color_map
```

A3. Quelle est la complexité de l'algorithme de coloration gloutonne dans le pire des cas ?

**Solution :** Dans le pire des cas, le graphe est complet. On obtient $O(n(n-1+c))$ si le nombre de couleurs dans la liste des couleurs est $c$.

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.

```python
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= greedy_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 courte 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 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, seed=7)
```

```python
    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

```python
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 :**

| $\Delta$ | 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 | 13 | 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 1). Cette fonction renvoie le dictionnaire des distances les plus courtes depuis le sommet `start` ainsi que le dictionnaire du parent sélectionné pour chaque sommet afin d'atteindre ces distances minimales.

**Solution :**

```python
def pq_dijkstra(g, start):  # use priorityqueue
    n = len(g)
    pq = queue.PriorityQueue()
    d = {v: math.inf for v in range(n)}
    d[start] = 0
    parents = {0: 0}
    pq.put((d[start], start))
    while pq.qsize() > 0:
        delta, s = pq.get()
        for v, dv in g[s]:
            if d[v] > delta + dv:
                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, a_to)` qui renvoie le chemin le plus court du sommet de départ au sommet `a_to`. Cette fonction utilise le dictionnaire parents qui a été créé lors de l'exécution de l'algorithme de Dijsktra.

**Solution :**

```
def build_path(parents, a_to):
    path = []
    while parents[a_to] != a_to:
        path.append(a_to)
        a_to = parents[a_to]
    path.append(a_to)
    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 peu être visualisé sur la figure **??**.

**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

**Fonction** DIJKSTRA($g$, $s_0$) ▷ Trouver les plus courts chemins à partir de $s_0$
    $n \leftarrow$ nombre de sommets de G
    pq $\leftarrow$ une file de priorités ▷ contient les tuples (distance, sommet)
    ENFILER(pq,(0,$s_0$)) ▷ initialisation de la file de priorités
    d $\leftarrow$ un dictionnaire ▷ recense les distances à $s_0$
    $\forall s \in S, d[s] \leftarrow w(s_0, s)$ ▷ $w(s_0, s) = +\infty$ si $s$ n'est pas voisin de $s_0$, 0 si $s = s_0$
    parents $\leftarrow \{s_0 : s_0\}$ ▷ parents[s] : le parent de s dans le plus court chemin de $s_0$ à s (dictionnaire)
    **tant que** pq n'est pas vide **répéter**
        $\delta, u \leftarrow$ DÉFILER(pq) ▷ Choix glouton !
        **pour** $v \in g[u]$ **répéter** ▷ Pour chaque voisin de $u$
            **si** $d[u] + \delta < d[v]$ **alors** ▷ si la distance est meilleure en passant par u
                $d[v] \leftarrow d[u] + \delta$ ▷ Mises à jour des distances des voisins
                ENFILER(pq, (d[v],v))
                parents[v] $\leftarrow u$ ▷ Pour garder la tracer du chemin le plus court
    **renvoyer** d, parents

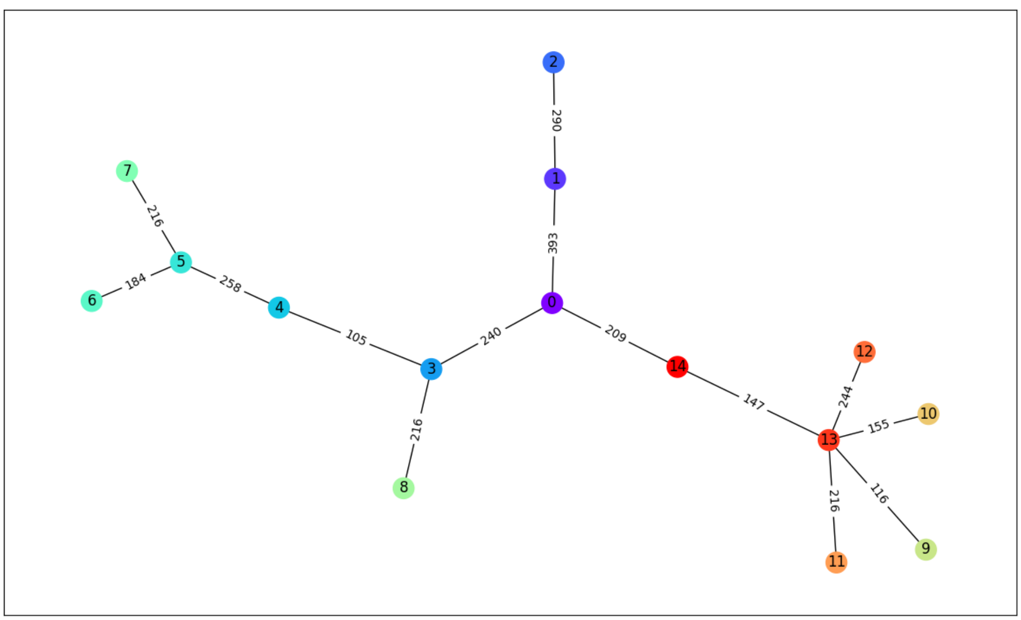


FIGURE 1 – Graphe issu de l'algorithme de Dijkstra : seules les arêtes sélectionnées par l'algorithme ont été conservées