Des tris, des couleurs et des distances

INFORMATIQUE COMMUNE - Devoir n° 4 - Olivier Reynet

Consignes :

1. utiliser une copie différente pour chaque partie du sujet,
2. écrire son nom sur chaque copie,
3. écrire de manière lisible et intelligible,
4. préparer une réponse au brouillon avant de la reporter sur la feuille.

A Trier conformément à un certain ordre

Cette section a pour objectif d'élaborer un code qui trie une liste conformément à l'ordre d'une autre liste.

■ **Exemple 1 — Trier conformément à.** Soit `K = [0, 1, 2, 3, 4, 5]` et `L=[3, 5, 2, 2, 4, 5]`. On souhaite trier dans l'ordre décroissant `L` et faire subir les mêmes modifications à la liste `K`.
Le tri décroissant de `L` résulte en `L = [5, 5, 4, 3, 2, 2]`, ce qui induit que `K` devient `[1, 5, 4, 0, 2, 3]`.

Dans ce but, on se propose d'utiliser le tri par insertion.

A1. On suppose qu'on dispose d'une liste **partiellement** triée `L` dans l'ordre décroissant jusqu'à l'indice `i-1`. Écrire une fonction de signature `insert_elem(L, i)` qui insère l'élément d'indice `i` à la bonne place dans `L[0 :i+1]`. Cette fonction agit en place et ne renvoie donc rien. Par exemple, si `L = [5,2,1,4,0,7]`, alors `insert_elem(L,3)` modifie `L` ainsi : `[5, 4, 2, 1, 0, 7]`.

Solution :

```
def insert_elem(L, i):
    to_insert = L[i]
    j = i
    while j > 0 and L[j - 1] < to_insert:
        L[j] = L[j - 1]
        j -= 1
    L[j] = to_insert
```

A2. Compléter le code de la fonction signature `tri_ins(L)` qui effectue le tri par insertion en place de la liste `L`.

```
def insertion_sort(t):
    for i in range(1, len(t)):
        # à compléter
```

Solution :

```
def insertion_sort(t):
    for i in range(1, len(t)):
        insert_elem(L, i)
```

A3. En vous inspirant du code précédent, écrire une fonction de signature `trier_conforme(L, K)` qui trie `K` conformément au tri de `L` dans l'ordre décroissant. Au début de la fonction, on garantira par une assertion que les deux listes passées en paramètre sont de même taille.

Solution :

```
def trier_conforme(L, K):
    n = len(L)
    m = len(K)
    assert n == m
    for i in range(1, n):
        to_insert_L = L[i]
        to_insert_K = K[i]
        j = i
        while j > 0 and L[j - 1] < to_insert_L:
            L[j] = L[j - 1]
            K[j] = K[j - 1]
            j -= 1
        L[j] = to_insert_L
        K[j] = to_insert_K
```

A4. Quelle est la complexité de la fonction `trier_conforme` dans le pire et le meilleur des cas ?

Solution : Cette complexité est celle du tri par insertion, car toutes les opérations ajoutées sont effectuées en tant constant $O(1)$. Donc $O(n)$ dans le meilleur des cas (la liste L est triée) et $O(n^2)$ dans le pire des cas (la liste L est triée dans le sens croissant). On en conclut que, ce tri conforme est codé à faible coût en terme de complexité ajoutée.

A5. En vous inspirant des questions précédentes, proposer une méthode pour trier conformément K selon L avec une complexité en $O(n \log n)$ dans tous les cas.

Solution : Il suffit d'utiliser le tri fusion en procédant de la même manière, c'est-à-dire en faisant subir à K les mêmes modifications qu'à L.

A6. Écrire un code qui implémente cette méthode et qui renvoie les deux listes. On décomposera ce code en deux fonctions récursives dont les signatures sont :

- `tcf(L: list[int], K: list[int])-> tuple(list[int], list[int])`,
- `f(L1: list[int], L2: list[int], K1: list[int], K2: list[int])-> tuple(list[int], list[int])`.

Solution :

```
def fusion(L1, L2, K1, K2):
    n1 = len(L1)
    n2 = len(L2)
    if n1 == 0:
        return L2, K2
    elif n2 == 0:
        return L1, K1
    else:
        if L1[0] > L2[0]:
            L, K = fusion(L1[1:], L2, K1[1:], K2)
            return [L1[0]] + L, [K1[0]] + K
        else:
            L, K = fusion(L1, L2[1:], K1, K2[1:])
            return [L2[0]] + L, [K2[0]] + K
```

```
def tcf(L, K):  # trier conforme fusion
    n = len(L)
    m = len(K)
    assert n == m
    if n < 2:
        return L, K
    else:
        L1, L2 = L[:n // 2], L[n // 2:]
        K1, K2 = K[:n // 2], K[n // 2:]
        tL1, tK1 = tcf(L1, K1)
        tL2, tK2 = tcf(L2, K2)
        return fusion(tL1, tL2, tK1, tK2)
```

B Coloration de graphe

■ **Définition 1 — Coloration valide.** Une coloration d'un graphe est valide lorsque deux sommets adjacents n'ont jamais la même couleur.

■ **Définition 2 — Nombre chromatique.** Le nombre chromatique d'un graphe G est le plus petit nombre de couleurs nécessaires pour obtenir une coloration valide de ce graphe. On le note généralement $\chi(G)$.

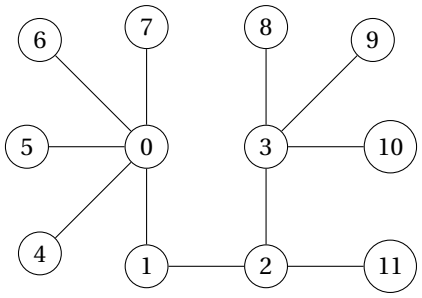


FIGURE 1 – Graphe `g`

On se donne un graphe `g` (cf. figure 1) ainsi qu'une liste de couleur utilisables pour colorer une graphe.

```
colors = ["pink", "turquoise", "yellow", "blue","magenta", "orange", "lime"]
```

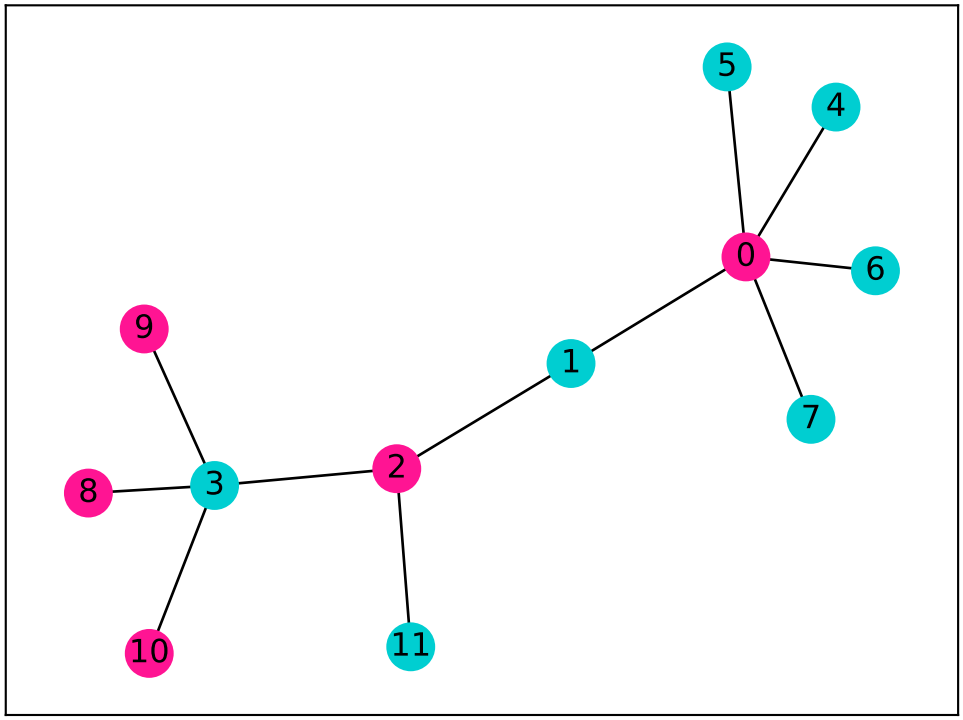
B7. En utilisant les listes Python, représenter par une liste d'adjacence le graphe `g` de la figure 1.

Solution :

```
g = [[1, 4, 5, 6, 7],
     [0, 2],
     [1, 3, 11],
     [2, 8, 9, 10],
     [0], [0], [0], [0],
     [3], [3], [3],
     [2] ]
```

B8. Quel est le nombre chromatique du graphe 1? Trouver ce nombre par le dessin et dessiner le graphe ainsi coloré.

Solution :
On colorie à la main le graphe. on voit que le graphe est planaire, donc on sait qu'il existe au moins une solution avec quatre couleurs. Peut-on cependant colorer avec trois ou même deux couleurs? Oui (cf. ci-dessous), donc $\chi_g = 2$.



L'algorithme de Welsh-Powell (cf. algorithme 1) est un algorithme de coloration de graphe glouton. Il traite les sommets dans l'ordre décroissant de leur degré. **Pour chaque sommet, dans l'ordre décroissant des degrés,** il choisit une nouvelle couleur et l'**affecte aux autres sommets du graphe si cela est possible**, c'est-à-dire s'ils ne sont ni déjà colorés **et** ni directement adjacents.

B9. Écrire une fonction de signature `degres(g)` dont le paramètre est un graphe sous la forme d'une liste d'adjacence. Cette fonction renvoie la liste des degrés des sommets du graphe. Par exemple, `[[1], [0, 2], [1, 3], [2]]` renvoie `[1, 2, 2, 1]`.

Solution :

```
def degres(g):
    n = len(g)
    d = []
    for i in range(n):
```

Algorithme 1 Welsh-Powell

```
1 : Fonction WP(g, couleurs)
2 :     n ← l'ordre du graphe g
3 :     sommets ← la liste des sommets du graphe triée dans l'ordre des degrés décroissants
4 :     cmap ← une liste à n éléments initialisés à None                ▷ les couleurs de chaque sommet
5 :     tant que sommets et couleurs ne sont pas vides répéter
6 :         s ← le sommet de degré le plus fort et le retirer de la liste
7 :         c ← une couleur de la liste couleurs et la retirer de la liste
8 :         cmap[s] ← c                                                 ▷ Colorer le sommet s en c
9 :         colorables ← liste des sommets colorables en c parmi les sommets restants dans l'ordre
10 :        pour chaque v de colorables répéter
11 :            cmap[v] ← c                                             ▷ Colorer le sommet v en c
12 :            supprimer v de la liste sommets
13 :    renvoyer cmap
```

```
            d.append((len(g[i])))
        return d
```

B10. Écrire une fonction de signature `remove(L: list[int], e)-> list[int]` qui renvoie une nouvelle liste copie de `L` mais dans laquelle toutes les occurrences de l'élément de valeur `e` ont été supprimées.

Solution :

```
def remove(L, e):
    S = []
    for elem in L:
        if elem != e:
            S.append(elem)
    return S
```

B11. Écrire une fonction de signature `colorables(g, s)-> list[bool]` qui prend comme paramètre le graphe `g` sous la forme d'une liste d'adjacence et `s` un sommet. Cette fonction renvoie une liste de booléens de la taille de `g` initialisées à `True`, sauf pour `s` et les voisins de `s`. Par exemple, `colorables([[1], [0, 2], [1, 3], [2]], 1)` renvoie `[False, False, False, True]`

Solution :

```
def colorables(g, s):
    n = len(g)
    c = [True for _ in range(n)]
    c[s] = False
    for u in g[s]: # les voisins de s ne sont pas colorables à l'identique
        c[u] = False
    return c
```

B12. Écrire une fonction de signature `meme_couleur(g, s, sommets)` qui renvoie la liste des sommets de `g` colorables de la **même** couleur que `s` parmi les sommets de la liste `sommets`. On utilisera la liste générée par la fonction `colorables`. Par exemple, `meme_couleur([[1], [0, 2], [1, 3], [2]],0,[2,3])` renvoie `[2]`.

Solution :

```
def meme_couleur(g, s, sommets):
    c = colorables(g,s)
    to_color = []
    for v in sommets: # dans l'ordre des degrés
        if c[v]:
            to_color.append(v)
            for k in g[v]:  # les voisins de v ne sont pas colorables à l'identique
                c[k] = False
    return to_color
```

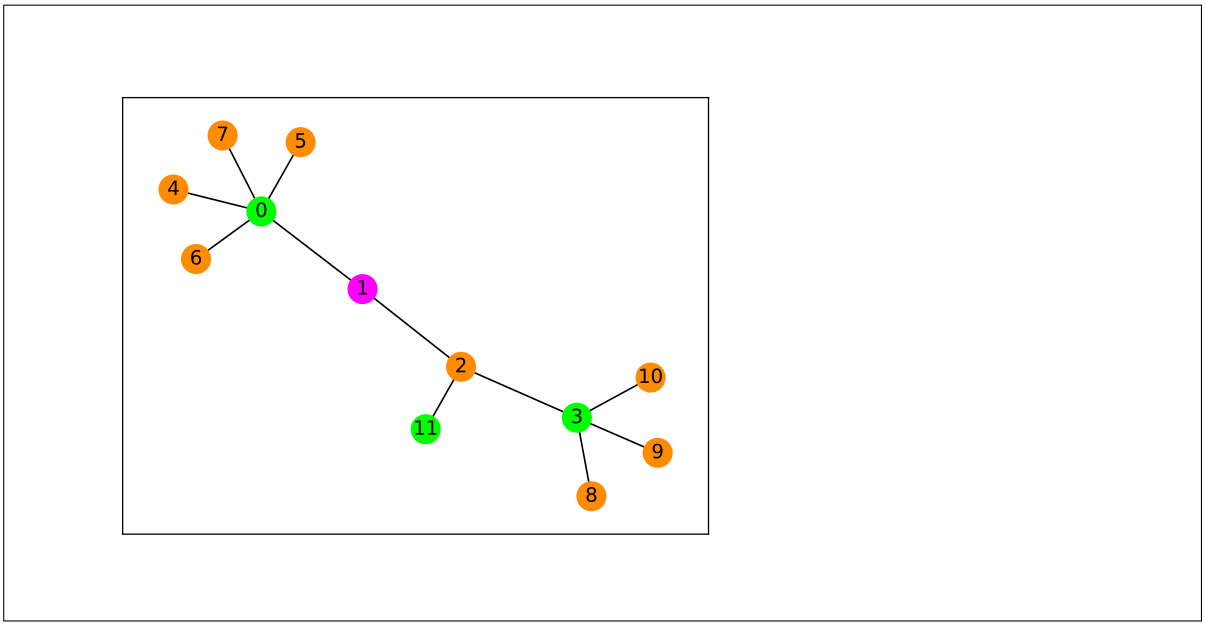
B13. Écrire une fonction de prototype `welsh_powell(g, colors)` qui implémente l'algorithme de Welsh-Powell. Cette fonction renvoie une liste `cmap` telle que `cmap[i]` est la couleur du sommet i. S'appuyer sur les fonctions précédentes.

Solution :

```
def welsh_powell(g, colors):
    n = len(g)
    deg = degres(g)
    sommets = [i for i in range(n)]
    color_map = [None for _ in range(n)]
    trier_conforme(deg, sommets)
    while len(sommets) > 0 and len(colors) > 0:
        s = sommets.pop(0)  # choix glouton
        c = colors.pop()
        color_map[s] = c  # coloration de s
        to_color = meme_couleur(g, s, sommets)
        for v in to_color:
            color_map[v] = c
            sommets = remove(sommets, v)
    return color_map
```

B14. Appliquer à la main l'algoritme de Welsh-Powell sur le graphe `g` de la figure 1. Que pouvez-vous en conclure ?

Solution : L'algorithme de Welsh-Powell n'est pas optimal. Sur cet exemple, on a besoin de trois couleurs alors que $\chi_g = 2$.



B15. Comment peut-on qualifier cet algorithme ?

Solution : C'est un algorithme glouton non optimal qui effectue le choix de l'optimal local (le sommet de degré le plus élevé) à chaque itération, sans se préoccuper de l'optimisation globale (utiliser le moins de couleur pour tout le graphe).

C Parcours et vérification

On souhaite vérifier que l'agorithme de coloration précédent a bien généré une coloration valide, c'est-à-dire que chaque sommet possède une couleur distincte de celle de ses voisins.

C16. On suppose que le graphe coloré est connexe et non orienté. Proposer un algorithme fondé sur le parcours en largeur pour effectuer cette vérification. Donner des explications sur la méthode en langage naturel, sans coder en Python.

Solution : On peut utiliser le parcours en largeur pour s'assurer de vérifier chaque sommet du graphe. Puis, lors de l'exploration des voisins d'un sommet, on peut ajouter un test pour vérifier que la couleur des voisins découverts n'est pas la même que celle du sommet en cours d'exploration. Si ce test échoue, la coloration n'est pas valide.

C17. Compléter le code ci-dessous afin d'implémenter cette vérification. Sur la copie, bien mettre en exergue les `#X` complétés.

```
def coloration_valide(g, color_map):
    n = len(g)
    F = []
    D = [False for _ in range(n)]
    F.append(0)
    D[0] = #1
    while len(F) > 0:
        s = #2
        for #3 in #4:
            if #5:
```

```
                return False
            if not D[v]:
                D[v] = True
                F.append(v)
    return #6
```

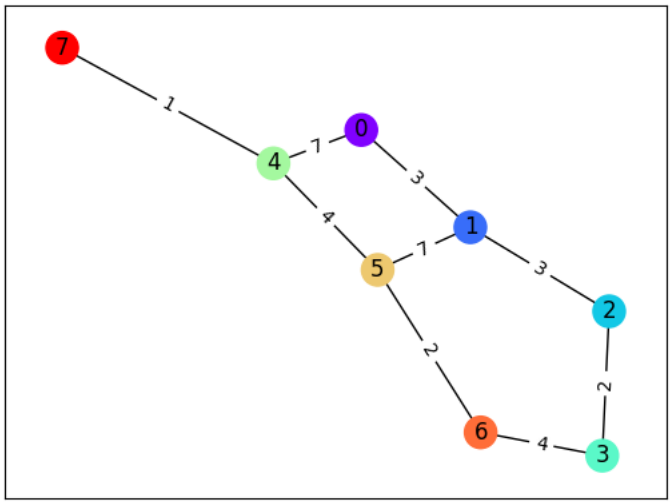
Solution :

```
def coloration_valide(g, color_map):
    n = len(g)
    F = []
    D = [False for _ in range(n)]
    F.append(0) # Le graphe est connexe et non orienté. Faut bien commencer quelque part !
    D[0] = True
    while len(F) > 0:
        s = F.pop(0)
        for v in g[s]:
            if color_map[v] == color_map[s]:
                return False
            if not D[v]:
                D[v] = True
                F.append(v)
    return True
```

On considère maintenant le graphe ci-dessous, donné sous la forme d'une matrice d'adjacence :

```
G = [[0, 3, 0, 0, 7, 0, 0, 0],
    [3, 0, 3, 0, 0, 7, 0, 0],
    [0, 3, 0, 2, 0, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 4, 0],
    [7, 0, 0, 0, 0, 4, 0, 1],
    [0, 7, 0, 0, 4, 0, 2, 0],
    [0, 0, 0, 4, 0, 2, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0]]
```

C18. Dessiner G.



Solution :

C19. Comment peut-on qualifier ce graphe ?

Solution : C'est un graphe non-orienté et pondéré sous forme de liste d'adjacence.

C20. Convertir `G` en une liste d'adjacence.

Solution :

```
[[(1, 3), (4, 7)],
    [(0, 3), (2, 3), (5, 7)],
    [(1, 3), (3, 2)],
    [(2, 2), (6, 4)],
    [(0, 7), (5, 4), (7, 1)],
    [(1, 7), (4, 4), (6, 2)],
    [(3, 4), (5, 2)],
    [(4, 1)]]
```

C21. Écrire une fonction de signature `madj_to_ladj(g)` qui convertit un graphe de type `G` donné par sa matrice d'adjacence en une liste d'adjacence.

Solution :

```
def madj_to_ladj(g):
    n = len(g)
    L = [[] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if g[i][j] != 0:
                L[i].append((j, g[i][j]))
    return L
```

On souhaite calculer les plus courts chemins entre des sommets de même couleur. Dans ce but, on applique l'algorithme de Dijsktra.

C22. Pourquoi peut-on appliquer l'algorithme de Dijkstra au graphe g?

Solution : On peut l'appliquer car toutes les pondérations du graphe sont positives.

C23. On applique l'algorithme à la main au départ du sommet 2 en construisant le tableau des distances à chaque itération de l'algorithme de Dijsktra. Compléter le tableau ci-dessous.

Δ	0	1	2	3	4	5	6	7	Prochain
{2}	∞	3	0	2	∞	∞	∞	∞	
{2,…}	…	…	0	…	…	…	…	…	
{2,…}	…	…	0	…	…	…	…	…	
{2,…}	…	…	0	…	…	…	…	…	
{2,…}	…	…	0	…	…	…	…	…	
{2,…}	…	…	0	…	…	…	…	…	
{2,…}	…	…	0	…	…	…	…	…	
{2,…}	…	…	0	…	…	…	…	…	

Solution :

Δ	0	1	2	3	4	5	6	7	Prochain
{2}	∞	3	0	2	∞	∞	∞	∞	3
{2,3}	∞	3	0	2	∞	∞	6	∞	1
{2,3,1}	6	3	0	2	∞	10	6	∞	0
{2,3,1,0}	6	3	0	2	13	10	6	∞	6
{2,3,1,0,6}	6	3	0	2	13	8	6	∞	5
{2,3,1,0,6,5}	6	3	0	2	12	8	6	∞	4
{2,3,1,0,6,5,4}	6	3	0	2	12	8	6	13	7
{2,3,1,0,6,5,4,7}	6	3	0	2	12	8	6	13	

C24. L'algorithme de Dijkstra permet d'obtenir la distance du chemin le plus court entre 2 et 7. Quel est ce chemin? Combien coûte-t-il? Quelle structure de données pourrait-on utiliser pour trouver ce chemin par l'algorihtme de Dijkstra?

Solution : Le chemin le plus court de 2 à 7 pèse 13. Il s'agit de : $2 \to 3 \to 6 \to 5 \to 4 \to 7$. Pour trouver ce chemin, on peut utiliser une structure parent sur le chemin pour garder la trace des sommets qu'il faut emprunter. (cf. cours)

On dispose du code qui implémente l'algorithme de Dijskstra et qui produit le tableau des plus courtes distances.

C25. Écrire une fonction de signature `s_pp_meme_couleur(depart, d, color_map)` qui renvoie le sommet de même couleur le plus proche de `depart` ainsi que la distance à ce sommet. Le paramètre `d` est le tableau final des distances issu de l'algorithme de Dijkstra invoqué à partir de `depart`. Le paramètre `color_map` est le résulat de la coloration de ce graphe, une liste de couleur propre à chaque sommet : `color_map[i]` renvoie la couleur du sommet i.

Solution :

```
def s_pp_meme_couleur(depart, d, color_map):
    n = len(d)
    dmin = math.inf
    smin = None
    for s in range(n):
```

```
            if s != depart and color_map[s] == color_map[depart] and d[s] < dmin:
                dmin = d[s]
                smin = s
        return smin, dmin
```

C26. La liste des distances `d` est relative à un graphe comptant 10 000 sommets. Chaque distance est codée sur 64 bits. Donner la taille de l'espace mémoire nécessaire pour stocker cette liste en Ko.

Solution : Chaque entier est codé sur 64 bits soit 8 octets. On a donc besoin de $8 \times N$ octets soit 80 000 octets, soit 80 Ko.