Arbre couvrant minimal

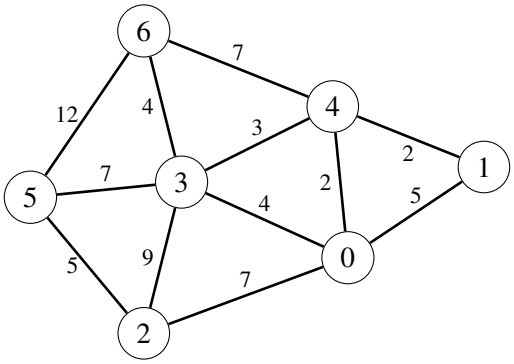
Devoir commun ITC 26/05/2023

MPSI - PCSI

Recommandations générales

- Dans tout le problème, seules les fonctions *usuelles* du langage python présentées en cours et en tp sont autorisées.
- Bien que la signature des fonctions n'apparaisse pas sur l'énoncé, celle-ci devra figurer sur votre copie accompagnée par un bref commentaire expliquant l'effet de la fonction. Par contre, sauf lorsque c'est explicitement demandé, vous n'avez pas à écrire une docstring complète, ni vérifier par la commande `assert` que les arguments de la fonction sont bien du type attendu.
- D'autre part, si les programmes les plus immédiats peuvent être donnés sans justifications, ceux plus délicats seront accompagnés de commentaires et/ou d'un (bref) texte d'explications.
- Les complexités demandées sont implicitement comprises dans le pire des cas, et on précisera la classe à laquelle appartient cette complexité ($O(n)$, $O(n^2)$, etc).

On considère dans tout le sujet un graphe **non orienté** et **pondéré**.
Par exemple :

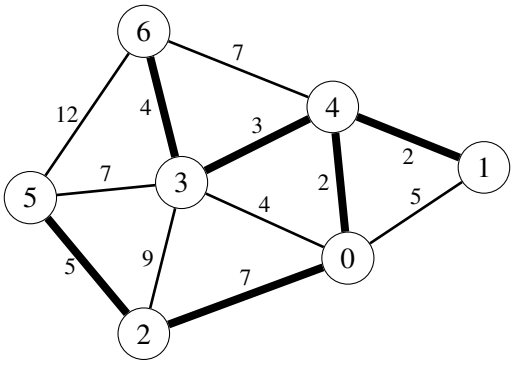


Graphe 1

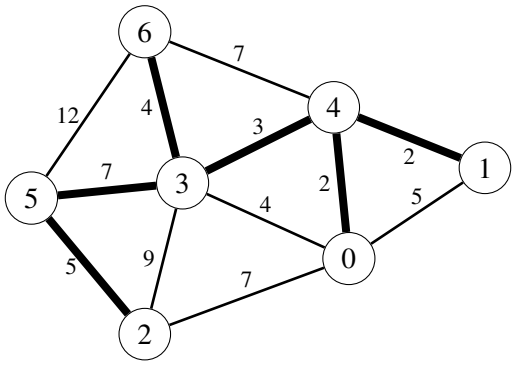
Le but est de construire un sous-graphe de ce graphe, en retirant certaines arêtes (on conserve tous les sommets), de manière à ce que :

- deux sommets reliés par un chemin dans le graphe initial doivent le rester dans le sous-graphe,
- et la somme des pondérations des arêtes retenues dans le sous-graphe doit être minimale.

Un sous-graphe vérifiant ces deux conditions est appelé un **arbre couvrant minimal**.
Par exemple, voici un arbre couvrant minimal du graphe précédent (les arêtes retenues sont celles mises en gras) :

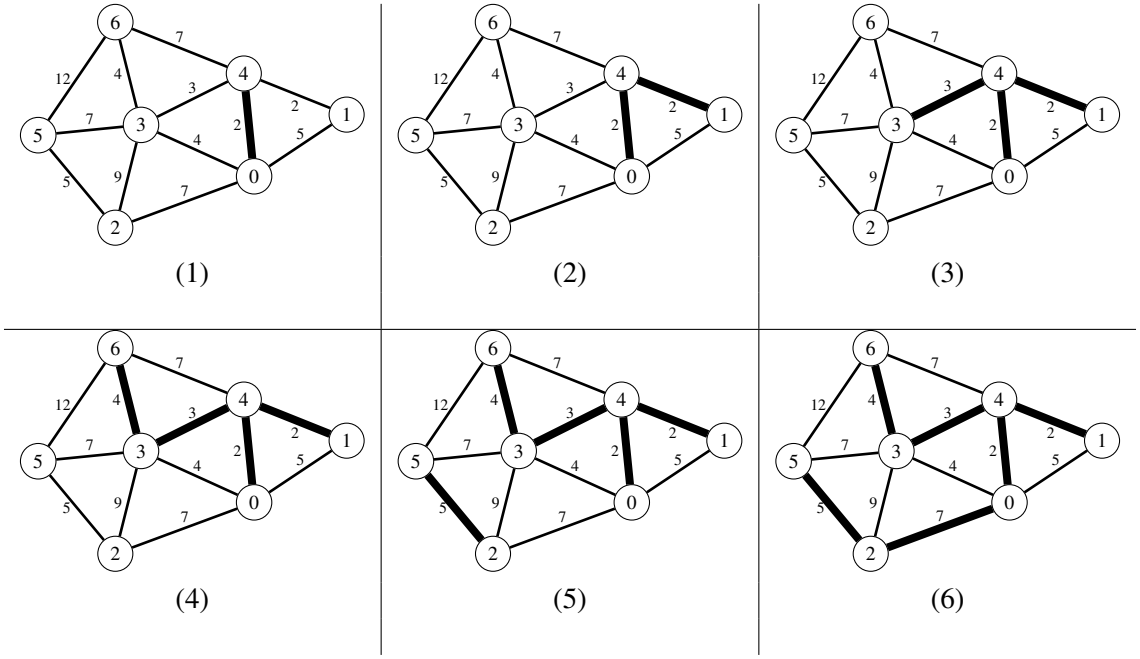


Comme le graphe choisi en exemple est connexe, la première contrainte signifie que le sous-graphe doit l'être aussi, ce qui est bien le cas, et on admet ici que le poids total de ce sous-graphe, qui vaut 4 + 5 + 3 + 2 + 7 + 2 = 23, est bien le plus petit que l'on puisse obtenir pour un tel sous-graphe connexe.
Notons qu'un arbre couvrant minimal n'est pas nécessairement unique puisqu'il peut arriver que l'on obtienne exactement le même poids total pour un autre choix d'arêtes.
C'est le cas du graphe en exemple où un autre arbre couvrant minimal est le suivant :



dont le poids total vaut ici aussi 4 + 3 + 2 + 2 + 7 + 5 = 23.

Le sujet se propose de programmer la construction d'un arbre couvrant minimal par l'algorithme dit de **Kruskal**.
Son principe est le suivant : on teste l'une après l'autre chacune des arêtes du graphe initial **dans l'ordre croissant de leurs poids**, et on l'ajoute au sous-graphe si et seulement si les deux sommets de l'arête considérée **ne sont pas déjà reliées par un chemin dans le sous-graphe en construction**.
Notons que si deux arêtes ont le même poids, elles peuvent être traitées indifféremment dans n'importe quel ordre (l'arbre minimal construit pourra alors être différent).
Dans le cas de notre exemple, les différentes étapes de la construction de l'arbre couvrant minimal sont les suivantes :

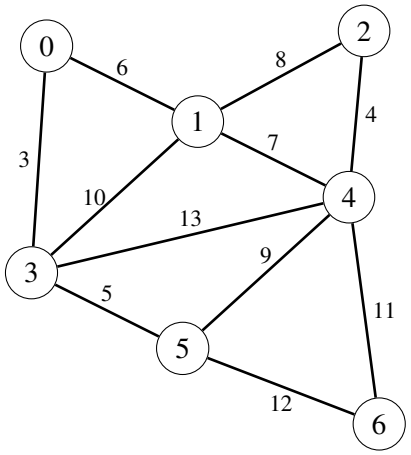


– **Étape (1) :** dans le graphe il y deux arêtes de poids minimal, égal à 2, qui sont $\{0, 4\}$ et $\{1, 4\}$. On part du principe ici que l'arête $\{0, 4\}$ est examinée en premier, et comme les sommets 0 et 4 qu'elle relie ne sont pas déjà reliés dans le sous-graphe en cours de construction (qui ne comporte pour l'instant aucune arête), on ajoute l'arête $\{0, 4\}$ au sous-graphe.

- **Étape (2) :** l'arête suivante à examiner est donc $\{1,4\}$ (elle aussi de poids 2). Comme les sommets 1 et 4 ne sont pas reliés dans le sous-graphe en cours de construction (qui ne comporte pour l'instant que l'arête $\{0,4\}$), on ajoute l'arête $\{1,4\}$ au sous-graphe.
- **Étape (3) :** l'arête suivante dans l'ordre croissant des poids est $\{3,4\}$ (de poids 3), dont les sommets ne sont pas déjà reliés dans le sous-graphe, donc on l'ajoute.
- **Étape (4) :** puis vient le tour des arêtes $\{0,3\}$ et $\{3,6\}$ (de poids 4), et on suppose que $\{0,3\}$ est traitée en premier. Ici, les sommets 0 et 3 sont déjà reliés dans le sous-graphe (par le chemin $0 \to 4 \to 3$), on ne l'ajoute donc pas au sous-graphe. On passe alors à l'arête $\{3,6\}$, que l'on ajoute puisque les sommets 3 et 6 au contraire ne sont pas reliés dans le sous-graphe.
- **Étape (5) :** puis on passe aux arêtes $\{0,1\}$ et $\{2,5\}$ (de poids 5). On suppose que $\{0,1\}$ est traitée en premier, et comme les sommets 0 et 1 sont déjà reliés dans le sous-graphe (par le chemin $0 \to 4 \to 1$), on ne l'ajoute pas. Ensuite, l'arête $\{2,5\}$ est ajoutée puisque ses sommets ne sont pas déjà reliés.
- **Étape (6) :** puis on passe aux trois arêtes (de poids 7) $\{0,2\}$, $\{3,5\}$ et $\{4,6\}$. On suppose que $\{0,2\}$ est traitée en premier, et comme les sommets 0 et 2 ne sont pas déjà reliés, on l'ajoute. Ensuite, on parcourt les deux autres arêtes de poids 7, puis toutes celles de poids supérieur, mais aucune n'est ajoutée puisque leurs sommets sont toujours reliées dans le sous-graphe. Ayant examiné toutes les arêtes, l'algorithme prend fin, et nous avons obtenu un arbre couvant minimal en ne conservant du graphe initial que les arêtes :

$$\{0,4\}, \{1,4\}, \{3,4\}, \{3,6\}, \{2,5\}, \{0,2\}.$$

Q1) Voici un autre exemple de graphe :



Graphe 2

Recopiez-le sur votre copie, et superposez l'arbre couvrant minimal construit à l'aide de l'algorithme de Kruskal (on tracera en rouge les arêtes retenues). On n'attend ici **aucune justification**.

On choisit de coder un graphe par un dictionnaire d'adjacence précisant pour un sommet donné la liste des sommets qui sont reliés ainsi que le poids de l'arête correspondante. Plus précisément, s'il y a dans le graphe une arête reliant un sommet a à un sommet b avec un poids p, alors la clé associée au sommet a a pour valeur une liste dans laquelle figurera le couple (b,p) (auquel s'ajoutent les autres couples correspondants aux arêtes dont a est l'une des deux extrémités). De même, la clé associée à b aura pour valeur une liste dans laquelle figurera le couple (a,p).

Par exemple, le dictionnaire associé à **Graphe** 1, présenté au début du sujet, est :

$$\begin{aligned} G = \{&0 : [(1,5),(2,7),(3,4),(4,2)],\\ &1 : [(0,5),(4,2)],\\ &2 : [(0,7),(3,9),(5,5)],\\ &3 : [(0,4),(2,9),(4,3),(5,7),(6,4)],\\ &4 : [(0,2),(1,2),(3,3),(6,7)],\\ &5 : [(2,5),(3,7),(6,12)],\\ &6 : [(3,4),(4,7),(5,12)]\}. \end{aligned}$$

On suppose en outre que les sommets du graphe sont désignés par des entiers consécutifs à partir de 0, comme dans les deux graphes précédents donnés en exemple.

Q2) On souhaite écrire une fonction `liste_arêtes` prenant en paramètre le dictionnaire d'adjacence d'un graphe et renvoyant la liste de toutes les arêtes du graphe, sous la forme d'un triplet (p, d, a) où p est le poids de l'arête, d son sommet de départ et a celui d'arrivée, avec la condition $d < a$ (ceci afin que, bien que le graphe ne soit pas orienté, chaque arête ne figure qu'une seule fois dans la liste).

Pour **Graphe 1** par exemple, on souhaite renvoyer la liste suivante (à l'ordre des triplets entre-eux près) :

$$[(5,0,1),(7,0,2),(4,0,3),(2,0,4),(2,1,4),(9,2,3),(5,2,5),(3,3,4),(7,3,5),(4,3,6),(7,4,6),(12,5,6)]$$

a) Le code suivant vous est donné :

```
1 def liste_arêtes(G) :
2     L=[]
3     for s in G :
4         for c in G[s] :
5             if ... :
6                 L.append(...)
7     return L
```

Recopiez les lignes 5 et 6 en les complétant pour que l'effet de la fonction soit bien celui attendu.

b) Justifier que, si on note e le nombre d'arêtes du graphe alors la complexité de l'appel `liste_arêtes(G)` est en $O(e)$ (on compte les arêtes **non orientées**, e est donc la taille de la liste L renvoyée par la fonction).

Q3) L'algorithme de Kruskal nécessite de pouvoir identifier pour chaque sommet du graphe à quelle composante connexe du sous-graphe en construction il appartient, afin de savoir si l'arête examinée à une étape donnée doit être retenue (si ses sommet appartiennent à des composantes connexes différentes) ou pas. Pour cela on propose une méthode simple, qui sera améliorée à la fin du sujet, consistant à définir un dictionnaire CC ayant pour clés les sommets du graphe et pour valeur correspondante le plus petit sommet appartenant à la même composante connexe.

Par exemple, pour **Graphe 1**, avant de commencer la construction le dictionnaire est initialisé à :

$$CC = \{0:0,\ 1:1,\ 2:2,\ 3:3,\ 4:4,\ 5:5,\ 6:6\}$$

(car tous les sommets appartiennent à des composantes connexes distinctes), puis à la fin de Étape (1) le dictionnaire est :

$$CC = \{0:0,\ 1:1,\ 2:2,\ 3:3,\ 4:0,\ 5:5,\ 6:6\}$$

(car l'arête $\{0,4\}$ a été ajoutée au sous-graphe, donc 0 et 4 appartiennent à la même composante connexe), et ainsi de suite pour chaque étape :

- Étape (2) : $CC = \{0:0,\ 1:0,\ 2:2,\ 3:3,\ 4:0,\ 5:5,\ 6:6\}$,
- Étape (3) : $CC = \{0:0,\ 1:0,\ 2:2,\ 3:0,\ 4:0,\ 5:5,\ 6:6\}$,

- Étape (4) : CC = $\{0:0,\ 1:0,\ 2:2,\ 3:0,\ 4:0,\ 5:5,\ 6:0\}$,
- Étape (5) : CC = $\{0:0,\ 1:0,\ 2:2,\ 3:0,\ 4:0,\ 5:2,\ 6:0\}$,
- Étape (6) : CC = $\{0:0,\ 1:0,\ 2:0,\ 3:0,\ 4:0,\ 5:0,\ 6:0\}$.

A chaque étape, pour savoir si deux sommets appartiennent à la même composante connexe, il suffit donc de lire les valeurs associées dans le dictionnaire et tester si elles sont égales.

a) Lister sur le même modèle les valeurs successives prises par ce dictionnaire dans le cas de la construction de l'arbre minimal de **Graphe 2** fait en question 2.

b) Écrire une fonction `init_comp(G)`, où G est le dictionnaire associé au graphe, et renvoyant le dictionnaire CC initialisé avec pour clés tous les sommets du graphe, et comme valeur associé le sommet lui-même.

c) Écrire une fonction `même_comp(CC,s1,s2)`, où $s1$ et $s2$ sont deux sommets du graphe, renvoyant le booléen `True` s'ils appartiennent à la même composante connexe définies par le dictionnaire CC.

Par exemple, si

$$CC = \{0:0,\ 1:1,\ 2:2,\ 3:2,\ 4:0,\ 5:5,\ 6:6\}$$

alors l'appel `même_comp(CC,2,4)` doit renvoyer `False` (puisque les valeurs stockées dans le dictionnaire aux clés 2 et 4 sont distinctes) et l'appel `même_comp(CC,2,3)` doit renvoyer `True` (puisqu'au contraire elles sont égales).

d) Écrire une fonction `fusionne_comp(CC,s1,s2)`, où $s1$ et $s2$ sont deux sommets du graphe appartenant à des composantes connexes différentes, et qui **modifie** le dictionnaire CC en faisant en sorte que $s1$ et $s2$ appartiennent à la même composante connexe (cette fonction ne spécifiera pas de valeur de retour - donc renverra `None`).

Par exemple, si

$$CC = \{0:0,\ 1:1,\ 2:2,\ 3:2,\ 4:0,\ 5:5,\ 6:6\}$$

alors après l'appel `fusionne_comp(CC,2,4)` on doit avoir

$$CC = \{0:0,\ 1:1,\ 2:0,\ 3:0,\ 4:0,\ 5:5,\ 6:6\}.$$

En effet, on a comparé les valeurs stockées dans les clés 2 et 4, à savoir 2 et 0, choisi la plus petite (donc 0), et remplacé pour toutes les clés la valeur 2 par la valeur 0.

e) On note v le nombre de sommets du graphe. Préciser les complexités des trois fonctions `init_comp`, `même_comp` et `fusionne_comp` en fonction de v.

Q4) L'objectif de cette partie est de trier par poids croissants les arêtes du graphe.

On considère une liste L dont tous les éléments sont d'un même type, sur lequel une relation d'ordre est définie par Python, et on suppose cette liste **triée dans l'ordre croissant**.

On définit la fonction suivante, où x désigne une valeur du même type que ceux des éléments de la liste :

```
1 def insère(L,x) :
2     k=len(L)
3     L.append(x)
4     while k>0 and L[k-1]>L[k] :
5         L[k],L[k-1]=L[k-1],L[k]
6         k=k-1
```

Notez-bien qu'il n'y a pas d'instruction `return`, ce n'est pas une erreur.

a) Détailler dans un tableau les valeurs prises par les variables k et L, au fur et à mesure des passages dans la boucle `while`, lors de l'appel `insère([2,4,6,8],3)`. Même question pour les appels `insère([2,4,6,8],1)` et `insère([2,4,6,8],9)`.

b) Plus généralement, quel est l'effet de l'appel `insère(L,x)` ? Vous rédigerez votre réponse en écrivant une docstring complète de la fonction `insère` précisant son effet (la signature ne fera pas figurer de type pour x), en fournissant pour exemples d'exécution les trois exemples de la question précédente.

Ajoutez en tête du corps de la fonction des commandes `assert` vérifiant que L est bien une liste formée d'éléments tous de même type que x (à noter que la liste L peut-être vide, auquel cas il n'y a bien sûr plus de contraintes sur le type de ses éléments).

c) Prouver la terminaison de la boucle `while` de la fonction `insère`.

d) On note $L = [\ell_0, \dots \ell_{n-1}]$ le contenu de la liste L **avant** l'appel de la fonction `insère`.

Donner sans justification l'expression de k_i (contenu de la variable k à la fin du i^e passage dans la boucle `while`) en fonction de n et i.

Puis prouver, toujours pour cette boucle while, l'invariant suivant :

$$\text{« } L_i = [\ell_0, \dots, \ell_{n-i-1}, x, \ell_{n-i}, \dots, \ell_{n-1}] \text{ ».}$$

(On notera que cette assertion dans le cas particulier où $i = 0$ signifie « $L_0 = [\ell_0, \dots, \ell_{n-1}, x]$ », et dans le cas particulier $i = n$ signifie « $L_n = [x, \ell_0, \dots, \ell_{n-1}]$ ».)

e) Terminer la preuve de la fonction `insère`.

f) Calculer la complexité de l'appel `insère(L,x)` (on choisit de compter une opération pour l'instruction `append`, et une opération pour le test > quelque soit le type des valeurs sur lequel il s'applique).

g) On souhaite enfin écrire une fonction `tri_croissant(L)` où L désigne toujours une liste dont tous les éléments sont d'un même type sur lequel une relation d'ordre est définie par Python, mais plus nécessairement rangée dans l'ordre croissant, et qui **renvoie** une **copie** rangée dans l'**ordre croissant** de cette liste.

Le code suivant vous est donné :

```
1 def tri_croissant(L):
2     Ltriée=[]
3     for a in L :
4         ...
5     return Ltriée
```

Préciser l'instruction à placer en ligne 4 pour que l'effet de la fonction soit bien celui attendu, et justifier que la complexité de l'appel `tri_croissant(L)` est en $O(e^2)$, où e désigne la taille de la liste L.

h) On rappelle qu'en Python la relation d'ordre définie sur les tuples correspond à l'**ordre lexicographique**. Justifier brièvement que l'appel `tri_croiss(L)` appliqué à la liste L construite par l'appel `liste_arêtes(G)`, définie en question 2, renvoie une copie de cette liste où les arêtes sont triées par **poids** croissants.

Par exemple pour **Graphe 1**, le résultat est la liste :

$$[(2,0,4),(2,1,4),(3,3,4),(4,0,3),(4,3,6),(5,0,1),(5,2,5),(7,0,2),(7,3,5),(7,4,6),(9,2,3),(12,5,6)].$$

Q5) a) En utilisant les fonctions précédentes, écrire une fonction `kruskal(G)`, où G est le dictionnaire d'adjacence du graphe, et renvoyant la liste des arêtes retenues dans l'arbre couvrant minimal construit par l'algorithme de Kruskal.

Par exemple, dans le cas de **Graphe 1** l'appel renverra la liste

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

(Pour l'ordre des arêtes proposé dans la liste donnée en question 4h.)

b) Préciser en fonction de e (nombre d'arêtes) et v (nombre de sommets) la complexité de l'appel `kruskal(G)`.

c) Écrire une fonction `construit_ACM(L)`, où L est la liste des arêtes correspondant à l'arbre couvrant minimal obtenue par l'appel `kruskal(G)`, et qui construit le dictionnaire d'adjacence du graphe correspondant à cet arbre couvrant minimal (c'est là encore un graphe non orienté).
Par exemple, l'appel `construit_ACM(L)` pour la liste

$$L = [(2,0,4),(2,1,4),(3,3,4),(4,3,6),(5,2,5),(7,0,2)]$$

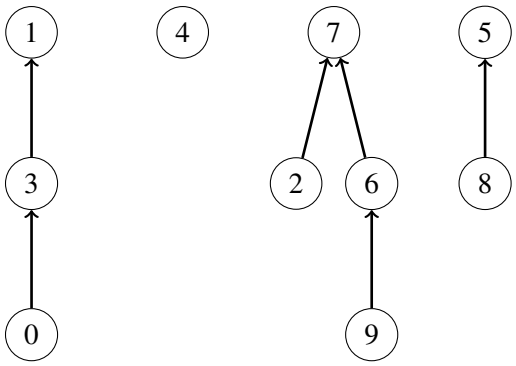
doit renvoyer le dictionnaire (à l'ordre près des arêtes dans la liste des voisins d'un sommet)

$$\begin{aligned}ACM = \{&0 : [(4,2),(2,7)],\\ &1 : [(4,2)],\\ &2 : [(5,5),(0,7)],\\ &3 : [(4,3),(6,4)],\\ &4 : [(0,2),(1,2),(3,3)],\\ &5 : [(2,5)],\\ &6 : [(3,4)]\}.\end{aligned}$$

Q6) Pour améliorer la complexité de l'appel `kruskal(G)`, il est possible d'utiliser un tri des arêtes plus rapide que le tri par insertion proposé en question 4 : les meilleurs tris sont en complexité $O(e\ln(e))$.
Un autre axe d'amélioration consiste à être plus performant sur la gestion des composantes connexes proposée en question 3, ce qui nous allons faire dans cette question en utilisant une structure de donnée appelée *union-find*.
Le principe est le suivant : on conserve une structure de dictionnaire `CC` dont les clés sont les sommets du graphe, mais au lieu de stocker une même valeur pour tous les sommets d'une même composante connexe du sous-graphe en construction, la valeur stockée pour un sommet est un autre sommet de la même composante connexe formant un chemin vers un unique sommet racine, auquel est associé la valeur `None`.
Par exemple, si un graphe a des sommets numérotés de 0 à 9 regroupés en composantes connexes de la manière suivante :

$$\{1,3,0\},\ \{4\},\ \{2,6,7,9\},\ \{5,8\}$$

alors on peut choisir définir les chemins :



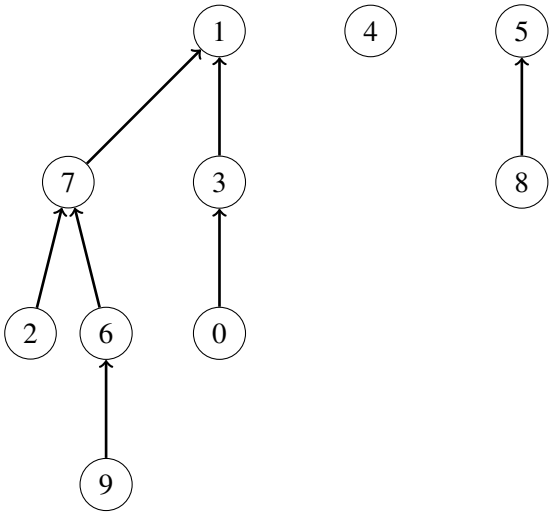
ce qui correspondra au dictionnaire :

$$CC = \{0:3,\ 1:\text{None},\ 2:7,\ 3:1,\ 4:\text{None},\ 5:\text{None},\ 6:7,\ 7:\text{None},\ 8:5,\ 9:6\}.$$

Avec cette représentation, la fusion de deux composantes connexes se fait très simplement en reliant la racine de l'une d'entre-elle (nous choisirons celle de plus petite valeur) à l'autre. Sur l'exemple précédent, pour fusionner les composantes connexes $\{1,3,0\}$ (de racine 1) et $\{2,6,7,9\}$ (de racine 7), la plus petite des deux racines étant 1 on modifie le dictionnaire en associant la valeur 1 à la clé 7 :

$$CC = \{0:3,\ 1:\text{None},\ 2:7,\ 3:1,\ 4:\text{None},\ 5:\text{None},\ 6:7,\ 7:1,\ 8:5,\ 9:6\}$$

ce qui correspond à la représentation suivante :

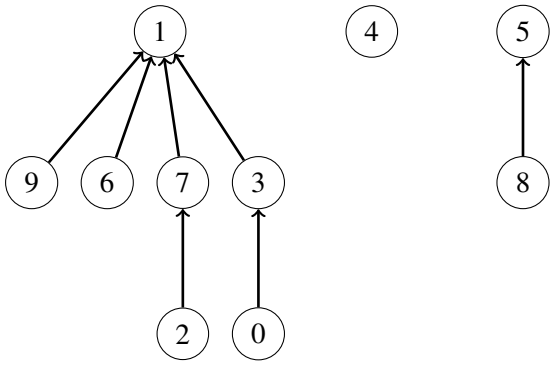


Composantes fusionnées

Pour déterminer si deux sommets appartiennent à la même composante connexe, il suffit pour chacun d'entre eux de « remonter » jusqu'à sa racine, et de vérifier si ces deux racines sont différentes.

a) Écrire les nouvelles versions des fonctions `init_comp(G)`, `même_comp(CC,s1,s2)` et `fusionne_comp(CC,s1,s2)`, présentées en question 3, utilisant ce nouveau principe.

b) La recherche de la racine associée à un sommet demande d'autant plus d'étapes que celui-ci est éloigné de la racine. Afin d'optimiser le temps de calcul, on peut procéder ainsi : pour chacun des sommets intermédiaires rencontrés lors de la recherche de la racine d'un sommet, on modifie la représentation de la composante connexe en lui associant directement la racine trouvée. Par exemple, pour la représentation obtenue sur la figure **composantes fusionnées**, si on cherche la racine associée au sommet 9, la « remontée » fait rencontrer successivement les sommets $9, 6, 7$ avant de trouver le sommet 1. On modifie alors la représentation en :



ce qui correspondra au dictionnaire :

$$\text{CC} = \{0 : 3,\ 1 : \text{None},\ 2 : 7,\ 3 : 1,\ 4 : \text{None},\ 5 : \text{None},\ 6 : 1,\ 7 : 1,\ 8 : 5,\ 9 : 1\}.$$

Écrire une nouvelle version de la fonction `même_comp(CC,s1,s2)` utilisant ce principe. Notez bien que si cette fonction continue à renvoyer uniquement un booléen, elle modifiera le dictionnaire CC passé en paramètre, cette modification étant à faire à la fois pour la recherche de la racine de $s1$ et celle de $s2$.