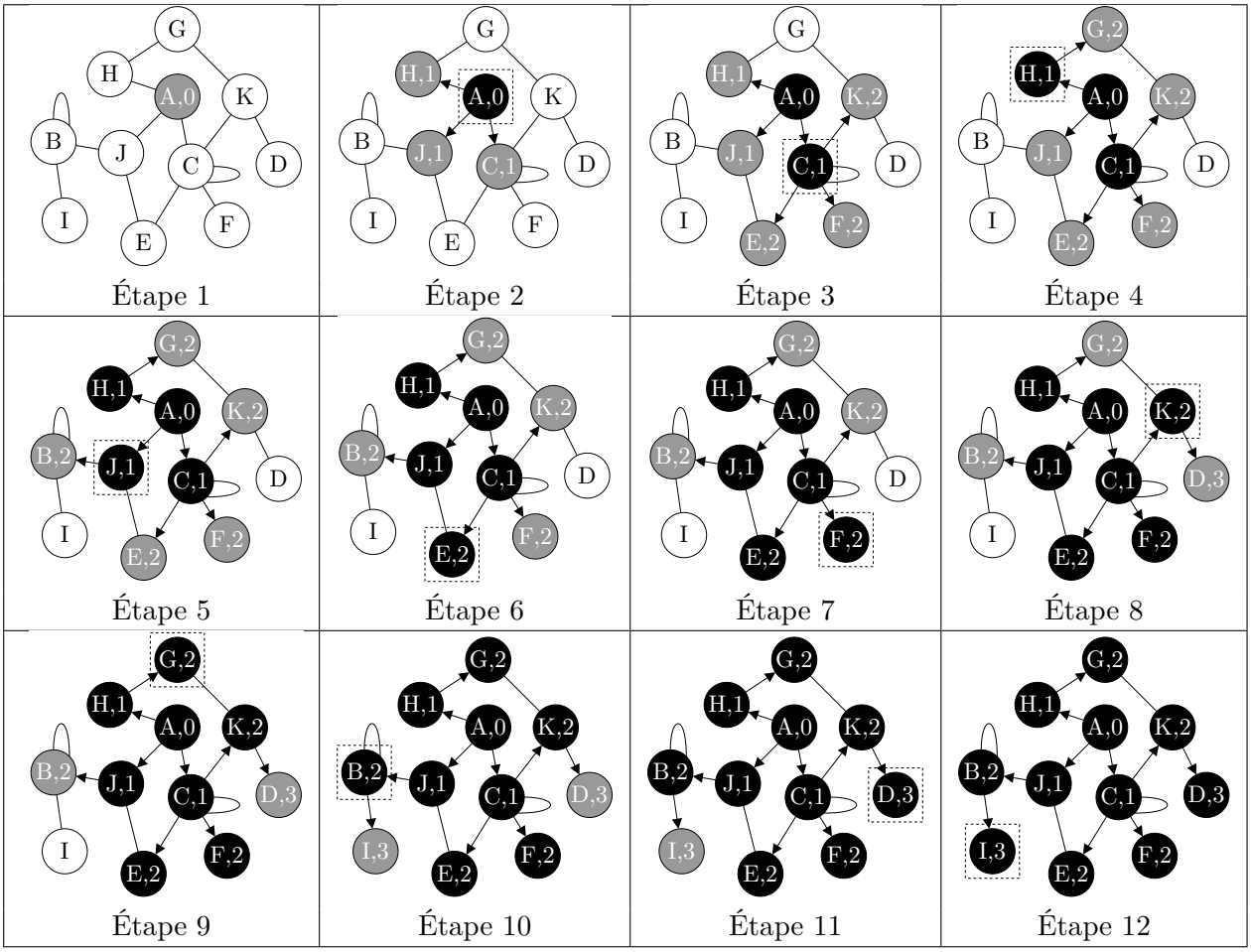
Exercice 1.

Question 1 – On utilise les notations du cours. Voici l'évolution des sommets blancs, gris et noirs :



	Étape 1	Étape 2	Étape 3	Étape 4	Étape 5	Étape 6
Noirs	∅	A	A, C	A, C, H	A, C, H, J	A, C, H, J, E
Gris	A	C, H, J	H, J, E, F, K	J, E, F, K, G	E, F, K, G, B	F, K, G, B

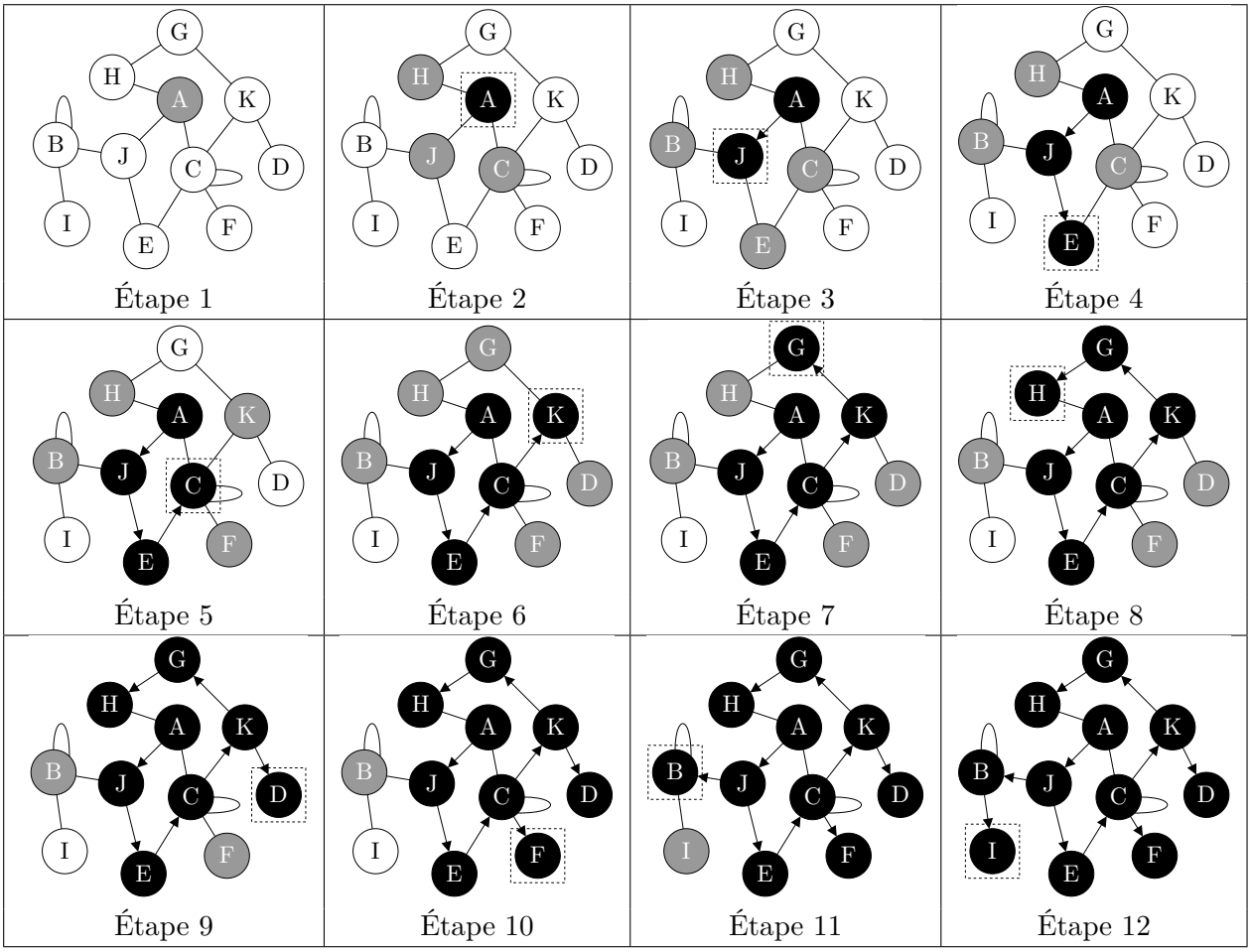
	Étape 7	Étape 8	Étape 9
Noirs	A, C, H, J, E, F	A, C, H, J, E, F, K	A, C, H, J, E, F, K, G
Gris	K, G, B	G, B, D	B, D

	Étape 10	Étape 11	Étape 12
Noirs	A, C, H, J, E, F, K, G, B	A, C, H, J, E, F, K, G, B, D	A, C, H, J, E, F, K, G, B, D, I
Gris	D, I	I	∅

Finalement, une des possibilités lors d'un parcours en largeur est de visiter les sommets dans l'ordre :

A, C, H, J, E, F, K, G, B, D, I

Question 2 – On utilise les notations du cours. Voici l'évolution des sommets blancs, gris et noirs :



	Étape 1	Étape 2	Étape 3	Étape 4	Étape 5	Étape 6
Noirs	∅	A	A, J	A, J, E	A, J, E, C	A, J, E, C, K
Gris	A	C, H, J	C, H, B, E	C, H, B, C	C, H, B, F, K	C, H, B, F, D, G

	Étape 7	Étape 8	Étape 9	Étape 10
Noirs	A, J, E, C, K, G	A, J, E, C, K, G, H	A, J, E, C, K, G, H, D	A, J, E, C, K, G, H, D, F
Gris	C, H, B, F, D, H	C, H, B, F, D	C, H, B, F	C, H, B

	Étape 11	Étape 12	
Noirs	A, J, E, C, K, G, H, D, F, B	A, J, E, C, K, G, H, D, F, B, I	A, J, E, C, K, G, H, D, F, B, I
Gris	C, H, I	C, H	∅

Finalement, une des possibilités lors d'un parcours en profondeur est de visiter les sommets dans l'ordre :

A, J, E, C, K, G, H, D, F, B, I

Exercice 2. Composantes connexes

Question 1 – Dans le graphe de la figure 1, il y a 4 composantes connexes :

$$S_1 = \{0\} \qquad S_2 = \{1, 13, 7, 8, 16, 14, 6\} \qquad S_3 = \{2, 3, 9, 10, 17, 19, 18\} \qquad S_4 = \{4, 5, 11, 12, 15\}$$

Question 2 – Soit $s \in S$ et $S' \subset S$ la composante connexe de s. Afin de déterminer S', on peut lancer un parcours en profondeur à partir de s ; l'ensemble S' contient alors tous les sommets visités lors de ce parcours. Ainsi, pour déterminer l'ensemble des composantes connexes d'un graphe, on peut utiliser la procédure suivante :

Entrées: Un graphe $G = (S, A)$.
Sortie: « `C: list[list[int]]` » la liste des composantes connexes de G.

`C` $\leftarrow$ `[]`
pour chaque sommet $s \in S$ **faire**
  si s n'a pas été visité **alors**
    On lance un parcours à partir de s.
    Soit « `compo: list[int]` » la liste des sommets de ce parcours.
    On ajoute `compo` à `C`.
  fin si
fin pour

Question 3.a – On écrit une fonction intermédiaire `parc_prof_aux` qui effectue une étape du parcours. Cette fonction prend entrée une liste « `V: list[int]` » contenant la liste des sommets visités jusqu'à présent ainsi qu'une liste « `dans_V: list[bool]` » telle que `dans_V[i]` vaut `True` si et seulement si `i` appartient à `V`.

```
def parc_prof_aux(M, s, N, dans_N):
    N.append(s)
    dans_N[s] = None
    for t in range(len(M)):
        if M[s][t] == 1 and not t in dans_N:
            parc_prof_aux(M, t, N, dans_N)
```

```
def parc_prof(M, s0):
    N = []; dans_N = {}
    parc_prof_aux(M, s0, N, dans_N)
    return N
```

Question 3.b –

```
def composantes_connexes(M):
    n = len(M)
    C = []
    dans_C = [False]*n
    for s0 in range(n):
        if not dans_C[s0]:
            compo = parc_prof(M, s0)
            C.append(compo)
            for s in compo:
                dans_C[s] = True
    return C
```

Question 3.c – Un graphe est connexe s'il y a 0 ou 1 composante connexe (0 composante connexe correspond à un graphe sans sommet).

```
def est_connexe(M):
    C = composantes_connexes(M)
    return len(C) <= 1
```

Exercice 3. Prédécesseurs dans un parcours de graphe

Question 1 – Notons que A n'a pas de prédécesseur puisque c'est le sommet initial.

Sommet	A	B	C	D	E	F	G	H	I	J	K
Prédécesseur	Aucun	J	A	K	C	C	H	A	B	A	C

Question 2 – Il suffit d'écrire un parcours en largeur dans lequel on construit le dictionnaire `P` décrit dans l'énoncé.

```
import collections
def parc_larg_ex3(d_G, s0):
    P = {}
    B = {s: True for s in d_G}  # Sommets blancs
    B[s0] = False
    G = collections.deque([s0]) # Sommets gris
    N = []                      # Sommets noirs
    while len(G) > 0:
        s = G.popleft()
        N.append(s)
        for t in d_G[s]:
            if B[t]:
                P[t] = s
                B[t] = False
                G.append(t)
    return P
```

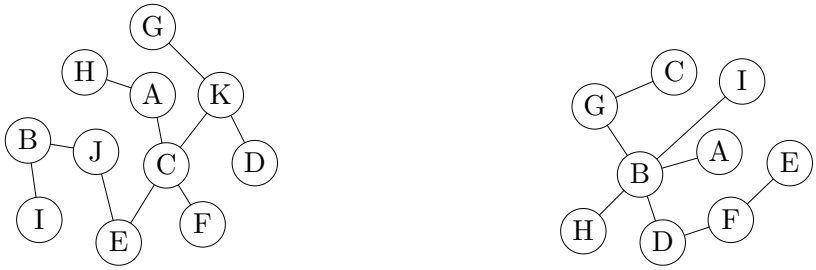
Question 3.a – On lance un parcours en largeur du graphe à partir du sommet t. Si s n'est pas visité lors de ce parcours, c'est qu'il n'y a pas de chaîne entre s et t. Sinon, on pose $u_0 = s$, on note u_1 le prédécesseur de u_0, puis u_2 le prédécesseur de u_1 et ainsi de suite. Il existe nécessairement $k \in \mathbb{N}$ tel que $u_k = t$ et alors $C = (u_0, u_1, \ldots, u_k)$ est une chaîne de s à t.

Question 3.b – Dans le cas où le parcours utilisé est un parcours en largeur, la chaîne est de longueur minimale, c'est à dire qu'elle est de taille $\text{dist}(s, t)$.

Question 3.c –

```
def find_path(d_G, s, t):
    P = parc_larg_ex3(d_G, t)
    C = [s]
    u = s
    while u != t:
        if u not in P:
            return None
        u = P[u]
        C.append(u)
    return C
```

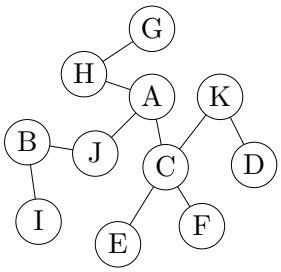
Question 4.a – Pour la figure 3, on peut par exemple prendre le graphe auquel on a supprimé les arêtes $\{A, J\}$, $\{G, H\}$, $\{B, B\}$ et $\{C, C\}$. Pour la figure 4, le graphe est déjà connexe sans cycle, donc l'arbre couvrant est le graphe lui-même :



Question 4.b – Si G n'est pas connexe, il n'existe pas d'arbre couvrant pour G. En effet, un graphe $G' = (S, A')$ tel que $A' \subset A$ ne peut pas être connexe.

Question 5.a – On lance un parcours en largeur avec un sommet initial arbitraire. Soit G' le graphe dont l'ensemble des sommets est S et tel que deux sommets s et t sont voisins si et seulement si s est le prédécesseur de t ou inversement. Alors G' est un arbre couvrant de G.

Question 5.b – On obtient l'arbre couvrant :



Question 5.c –

```
# Suppose que le graphe en entrée est connexe.
def make_AC(d_G, s0):
    P = parc_larg_ex3(d_G, s0)
    d_AC = {s: [] for s in d_G}
    for s in P:
        t = P[s]
        d_AC[s].append(t)
        d_AC[t].append(s)
    return d_AC
```

Question 6.a – Lorsqu'on lance un parcours en largeur à partir d'un sommet s_0 comme dans la question 5.c, on obtient un arbre couvrant de la composante connexe de s_0. Si on relance le parcours en largeur à partir d'un sommet s_1 qui n'est pas dans la composante connexe de s_0, on obtient un arbre couvrant de la composante connexe de s_1. On recommence ainsi jusqu'à ce que tous les sommets aient été visités. La réunion de tous les arbres couvrants obtenus est alors le graphe G' demandé par l'énoncé. En résumé, on obtient une procédure similaire à celle de l'exercice 2 :

- Soit « `d_Gp: dict[t: list[t]]` » un dictionnaire initialement vide. À la fin de la procédure, ce dictionnaire contiendra les listes d'adjacence de G'.
- Tant qu'au moins un sommet n'a pas encore été visité :
  - Soit s l'un des sommets n'ayant pas encore été visité.

- On lance un parcours en largeur à partir de s ce qui fournit un arbre couvrant de la composante connexe de s. On note « `d_AC: dict[t: list[t]]` » le dictionnaire représentant cet arbre couvrant.
- On ajoute à `d_Gp` tous les sommets et toutes les arêtes de `d_AC`. Notons qu'avant cette étape, aucun sommet de `d_AC` ne peut être présent dans `d_Gp`.

- On renvoie `d_Gp`.

Question 6.b – Il y a deux cas :

- Si $G = G'$, alors G ne contient pas de cycle (car G' est sans cycle).
- Si $G \neq G'$, soit $\{s_1, s_2\}$ une arête de G qui n'appartient pas à G'. Comme s_1 et s_2 sont dans la même composante connexe dans G, ces sommets sont aussi dans la même composante connexe dans G'. À l'aide d'un parcours dans G', on peut obtenir une chaîne élémentaire de s_2 à s_1 (voir question 3.c). En y ajoutant l'arête $\{s_1, s_2\}$, on obtient un cycle élémentaire de G.

Remarque. En réalité, on pourrait se passer du parcours dans le graphe G'. En effet, lors de la construction de `d_Gp`, les sommets s_1 et s_2 ont été ajoutés à `d_Gp` en même temps (à la suite d'un parcours de G). Si on note s_0 le sommet initial de ce parcours, on obtient une chaîne C_1 de s_1 à s_0 et une chaîne C_2 de s_2 à s_0 (comme dans la question 3.c). En combinant C_1, C_2 et l'arête $\{s_1, s_2\}$, on obtient une chaîne de G entre s_1 et lui-même. Attention, cette chaîne peut ne pas être un cycle élémentaire (si C_1 et C_2 se chevauchent), il faut donc en extraire un cycle élémentaire. Cette procédure est plus complexe que celle donnée dans la correction, mais elle est plus rapide.

Question 6.c –

```
def make_Gp(d_G):
    d_Gp = {}
    for s0 in d_G:
        if s0 not in d_Gp:
            P = parc_larg_ex3(d_G, s0)
            for s in list(P.keys()) + [s0]:
                if s in d_Gp:
                    raise RuntimeError("Cette ligne ne devrait pas être exécutée")
                d_Gp[s] = []
            for s in P:
                t = P[s]
                d_Gp[s].append(t)
                d_Gp[t].append(s)
    return d_Gp
```

```
def diff_aretes(d_G1, d_G2):
    """
    Renvoie la liste des arêtes qui sont dans d_G1, mais pas dans d_G2
    """
    A = {}
    for s in d_G2:
        for t in d_G2[s]:
            A[(s,t)] = None
            A[(t,s)] = None
    L = []
    for s in d_G1:
        for t in d_G1[s]:
            if (s,t) not in A:
                L.append((s,t))
                A[(s,t)] = None
                A[(t,s)] = None
    return L
```

```
def find_multiple_cycle(d_G):
    """
    Renvoie une liste de cycles (un cycle pour chaque arête de G qui n'est pas
    dans G').
    """
    d_Gp = make_Gp(d_G)
    L = diff_aretes(d_G, d_Gp)
    C = []
    for (s,t) in L:
        C.append(find_path(d_Gp, s, t))
        C[-1].append(s)
    return C
```

```
def find_cycle(d_G):
    C = find_multiple_cycle(d_G)
    if len(C) == 0:
        return None
    else:
        return C[0]
```

Exercice 4. Coloration d'un graphe

Question 1 – Les couleurs seront notées $1, 2, 3, \ldots, k$. Une coloration est donnée par un dictionnaire « `couleurs: dict[t: int]` » dont les clés sont les sommets et les valeurs sont les couleurs. Lorsqu'un sommet `s` n'a pas encore de couleur, on a `couleurs[s] = 0`.

```
def existe_boucle(d):
    for s in d:
        if s in d[s]:
            return True
    return False
```

```
def coloration_aux(d_G, couleurs, k, S, i):
    """
    d_G: dict[t: list[t]]
    couleurs: dict[t: int]
    k: int     --- Le nombre de couleurs qu'on peut utiliser.
    S: list[t] --- La liste des sommets.
    i: int     --- S[i] est le sommet qu'on essaye de colorer.
    Return: un couple (couleurs,k) qui colore le graphe.
            None si on ne peut pas colorer le graphe.
    ----------------
    Hypothèse: le graphe ne contient pas de boucle.
    """
    if i >= len(S):
        return couleurs, k
    s = S[i]
    couleurs_possibles = [True]*(k+1)
    for s2 in d_G[s]:
        couleurs_possibles[couleurs[s2]] = False
    for c in range(1, k+1):
        if couleurs_possibles[c]:
            couleurs[s] = c
            res = coloration_aux(d_G, couleurs, k, S, i+1)
            if res is not None:  # res != None
                return res
            else:
                couleurs[s] = 0
    return None
```

```
def coloration(d_G):
    if existe_boucle(d_G):
        return None
    S = [s for s in d_G]
    couleurs = {s: 0 for s in d_G}
    for k in range(1,len(d_G)+1):
        # print(k)
        res = coloration_aux(d_G, couleurs, k, S, 0)
        if res is not None:  # res != None
            return res
    print("Cette ligne ne devrait jamais être exécutée")
```