

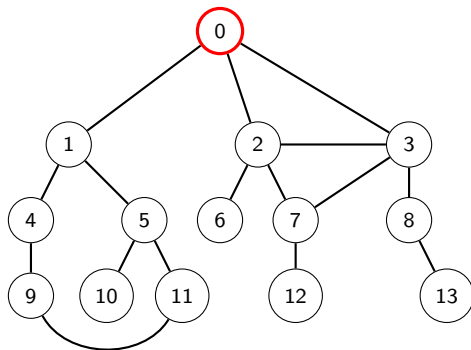
Informatique et Calcul Scientifique

Cours 12 : Algorithmes de graphes II

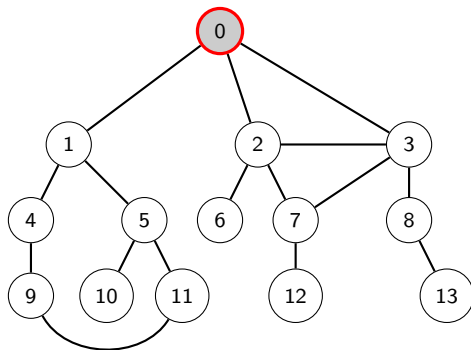
- ▶ Graphes : définition
 - ▶ graphes dirigés, graphes non dirigés
 - ▶ graphes pondérés
- ▶ Représentations de graphes par matrice d'adjacence et listes d'adjacence
- ▶ Chemin, plus court chemin, distance
- ▶ Parcours de graphes : en largeur (BFS) et en profondeur (DFS).

- ▶ Pour un graphe G et un sommet s de G , $\text{BFS}(G, s)$ parcourt exactement une fois chaque sommet de G atteignable depuis s .
- ▶ De plus, l'ordre de parcours produit par $\text{BFS}(G, s)$ satisfait la propriété suivante : pour tout $d \in \mathbb{N}^*$, tous les sommets à distance d de s sont parcourus avant n'importe quel sommet à distance $d + 1$ de s .

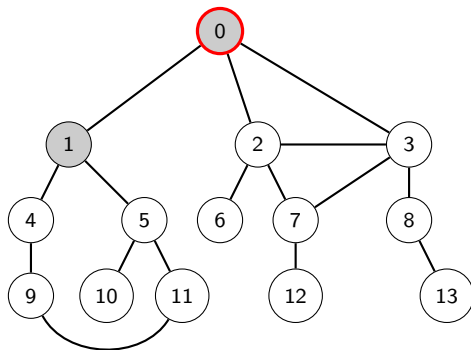
```
def BFS(G,s):  
    '''  
    Entree: graphe G en dict de listes d'adj, s sommet  
    Parcourt G en largeur, affiche les sommets parcourus  
    '''  
    n = len(G)  
    a_parcourir = [s]  
    vu = [0 for i in range(n)]  
    vu[s] = 1  
  
    while a_parcourir:  
        sommet = a_parcourir.pop(0)  
        for u in G[sommet]:  
            if not vu[u]:  
                a_parcourir.append(u)  
                vu[u] = 1  
        print(sommet)
```



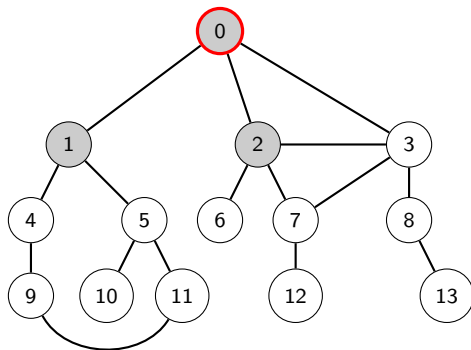
`a_parcourir = [0]`



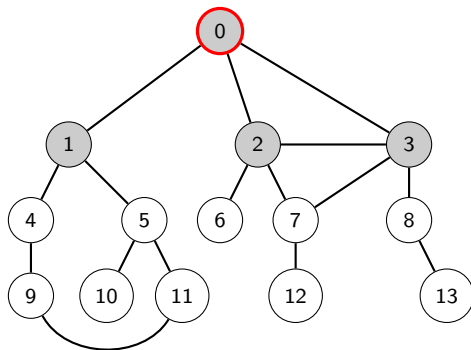
a_parcourir = [1, 2, 3]



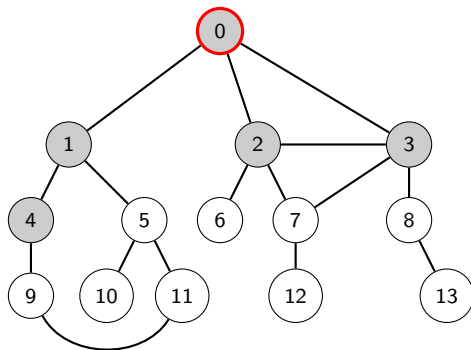
a_parcourir = [2, 3, 4, 5]



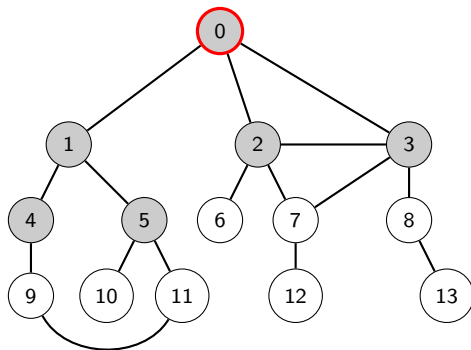
a_parcourir = [3, 4, 5, 6, 7]



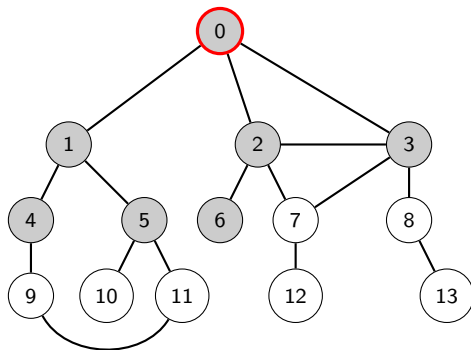
a_parcourir = [4, 5, 6, 7, 8]



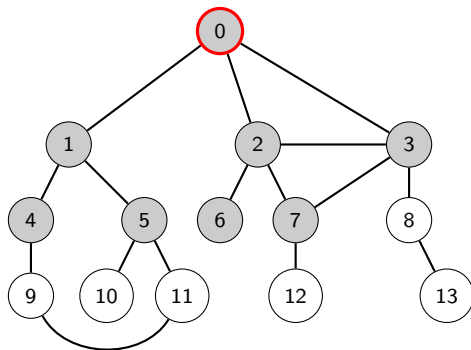
a_parcourir = [5, 6, 7, 8, 9]



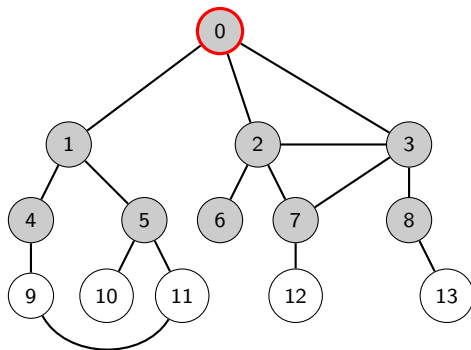
a_parcourir = [6, 7, 8, 9, 10, 11]



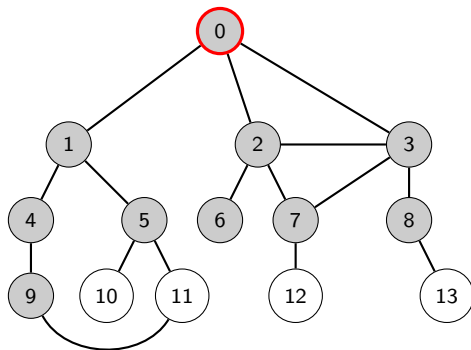
a_parcourir = [7, 8, 9, 10, 11]



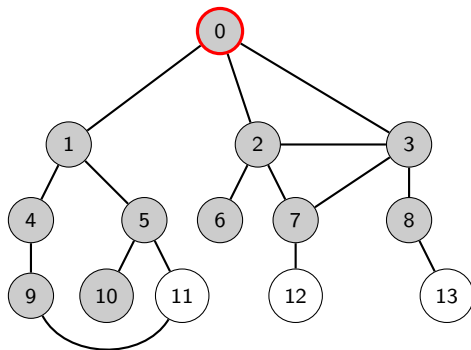
`a_parcourir = [8, 9, 10, 11, 12]`



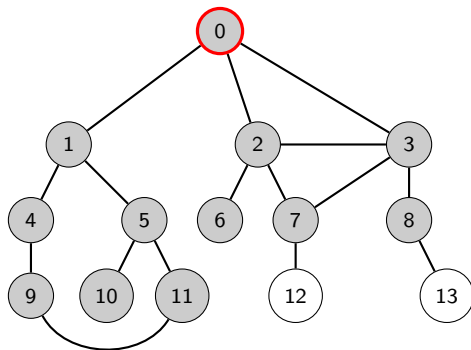
`a_parcourir = [9, 10, 11, 12, 13]`



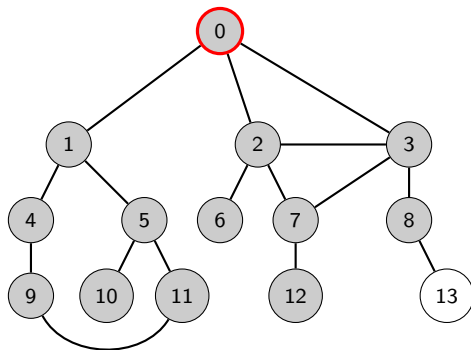
a_parcourir = [**10**, 11, 12, 13]



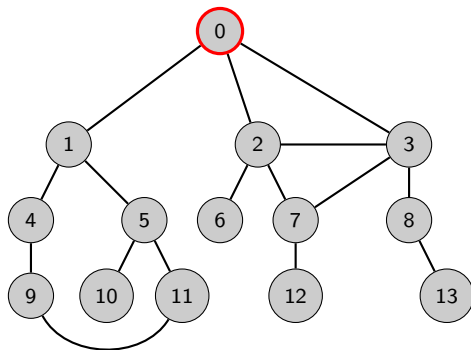
a_parcourir = [11, 12, 13]



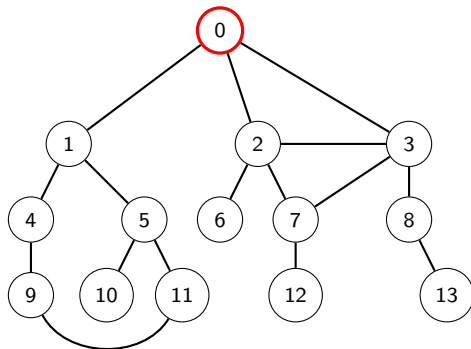
a_parcourir = [12, 13]

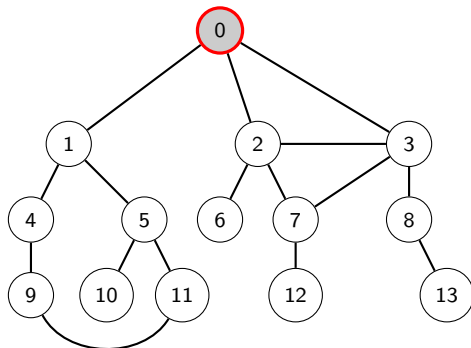


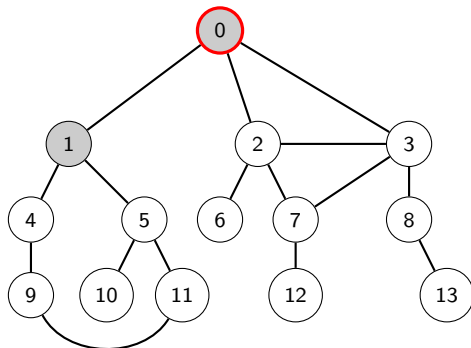
a_parcourir = [13]

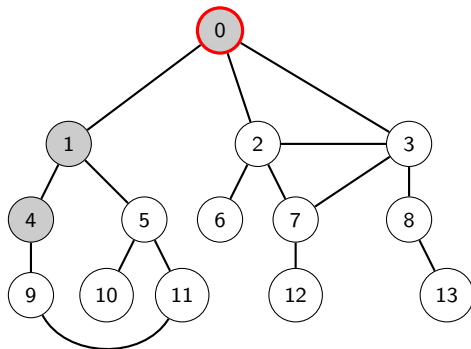


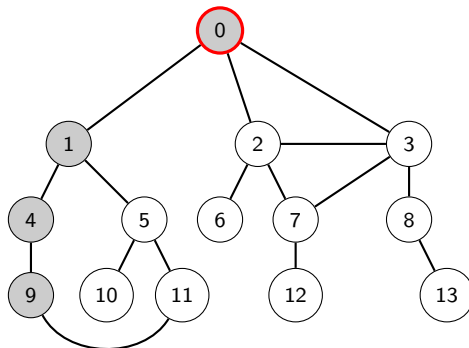
```
a_parcourir = []
```

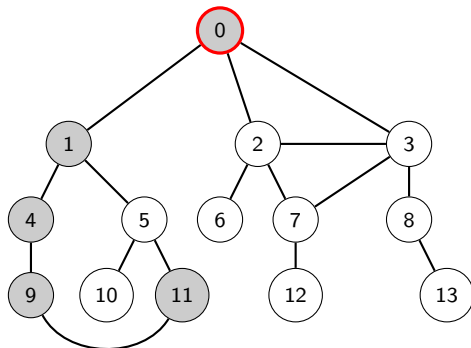


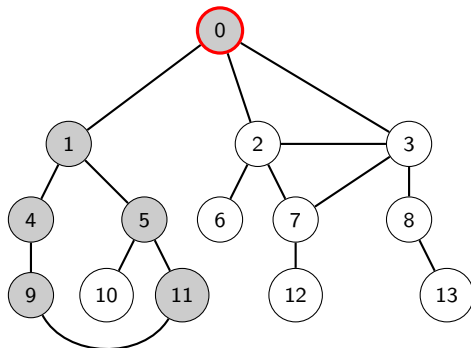


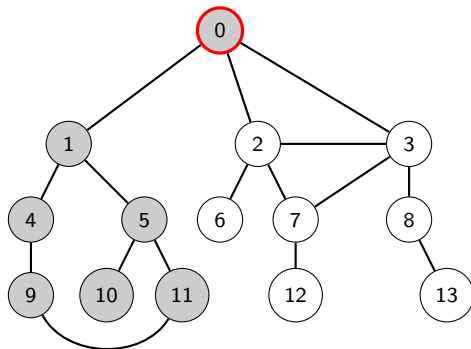


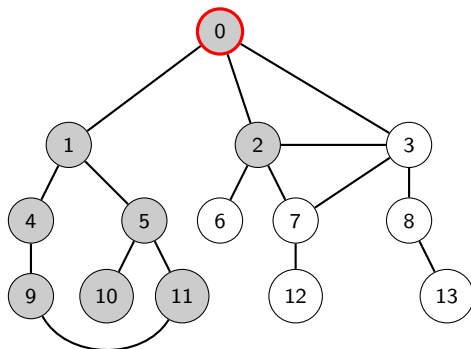


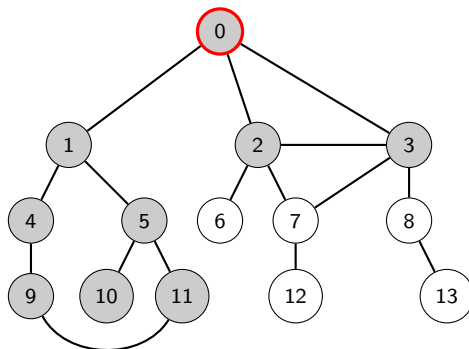


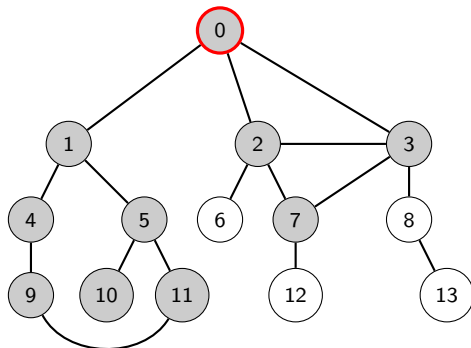


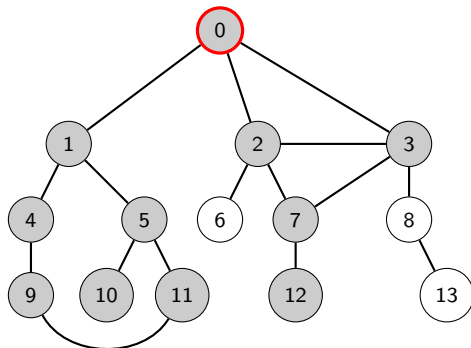


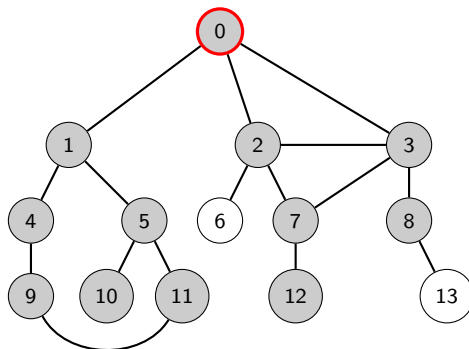


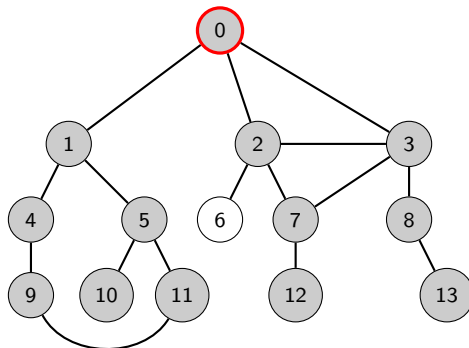


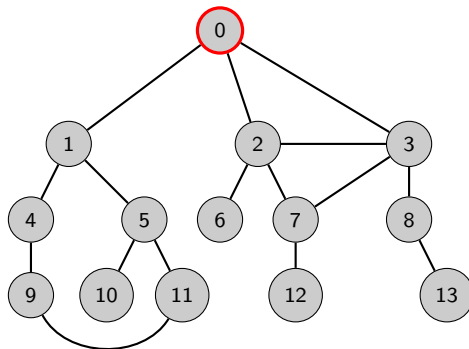












- ▶ Le parcours en profondeur est naturellement récursif : pour aller “le plus loin possible” dans une “branche” depuis un certain sommet avant d’aller dans une autre branche, il faut choisir un voisin de ce sommet et aller de nouveau “le plus loin possible” depuis ce sommet.
- ▶ L’algorithme peut être transformé en algorithme itératif en simulant le parcours récursif à l’aide d’une **pile** (qu’on verra un peu plus loin).

```
def DFS(G, s):  
    '''  
    Entree: graphe G en dict de listes d'adj, s sommet  
    Parcourt G recursivement en profondeur depuis s  
    '''  
    print(s)  
    vu[s] = 1  
    for u in G[s]:  
        if not vu[u]:  
            DFS(G,u)  
  
G = {0:[1,2], 1:[0,2], 2:[0,1]}  
vu = [0 for u in G]  
DFS(G, 0)
```

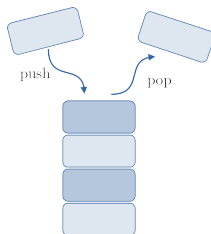
Structures de données pour BFS et DFS

- ▶ BFS maintient un ensemble (`a_parcourir`) de sommets à parcourir et choisit de cet ensemble un prochain sommet à parcourir à chaque étape.
- ▶ De même, la version itérative de DFS maintient un ensemble (`a_parcourir`) de sommets à parcourir, et la version récursive maintient implicitement un tel ensemble¹.
- ▶ La différence entre BFS et DFS est le choix du prochain sommet à parcourir : DFS choisit le dernier sommet qui a été ajouté à l'ensemble de sommets à parcourir, et BFS choisit le plus "ancien".
- ▶ Pour que ces algorithmes fonctionnent le plus efficacement possible, il faut choisir la bonne structure de données pour représenter l'ensemble de sommets à parcourir.

1. via la pile d'appels récursifs (recursion stack)

Pile (stack)

- ▶ Une **pile** (stack) est une structure de données qui permet d'effectuer efficacement (en temps constant) les opérations suivantes :
 - ▶ L'ajout d'un élément au "haut" de la pile (**push**)
 - ▶ L'accès à et la suppression de l'élément qui est actuellement au "haut" de la pile (**pop**)
- ▶ Penser à une pile de plateaux à la cafétéria...
- ▶ Une pile est une structure de données LIFO (Last In, First Out).



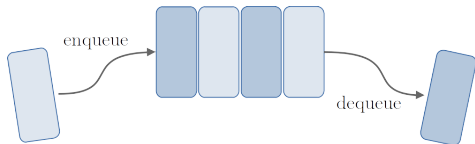
Pile (stack)

- ▶ Une **pile** (stack) est une structure de données qui permet d'effectuer efficacement (en temps constant) les opérations suivantes :
 - ▶ L'ajout d'un élément au "haut" de la pile (**push**)
 - ▶ L'accès à et la suppression de l'élément qui est actuellement au "haut" de la pile (**pop**)
- ▶ Penser à une pile de plateaux à la cafétéria...
- ▶ Une pile est une structure de données LIFO (Last In, First Out).
- ▶ La version itérative de DFS maintient l'ensemble des prochains sommets à explorer dans une pile².

2. Et la version récursive maintient une pile d'appels récursifs.

File (queue)

- ▶ Une **file** ou **file d'attente** (queue) est une structure de données qui permet d'effectuer efficacement (en temps constant) les opérations suivantes :
 - ▶ L'ajout d'un élément à la "fin" de la file (**enqueue**)
 - ▶ L'accès à et la suppression de l'élément qui est actuellement en "tête" de file (**dequeue**)
- ▶ Toujours à la cafétéria, penser aux gens qui font la queue.
- ▶ Une file est une structure de données FIFO (First In, First Out).



- ▶ Une **file** ou **file d'attente** (queue) est une structure de données qui permet d'effectuer efficacement (en temps constant) les opérations suivantes :
 - ▶ L'ajout d'un élément à la "fin" de la file (**enqueue**)
 - ▶ L'accès à et la suppression de l'élément qui est actuellement en "tête" de file (**dequeue**)
- ▶ Toujours à la cafétéria, penser aux gens qui font la queue.
- ▶ Une file est une structure de données FIFO (First In, First Out).
- ▶ BFS maintient l'ensemble des prochains sommets à explorer dans une file, ce qui garantit que pour $d \in \mathbb{N}^*$, tous les sommets à distance d de s sont parcourus avant n'importe quel sommet à distance $d + 1$ de s .

- ▶ En Python une **pile** est simplement implémentée par une liste :
 - ▶ On ajoute un élément au haut de la pile (i.e., en fin de liste) avec `L.append()`
 - ▶ On accède à l'élément au haut de la pile et on le supprime avec `L.pop()`
 - ▶ Ces deux opérations se font en temps constant.
- ▶ Par contre, on ne peut pas implémenter de manière efficace une **file** avec une liste car la suppression d'un élément en tête de file (i.e., en début de liste) prend temps $\Theta(n)$ pour une file de longueur n ...
- ▶ Pour implémenter une file en Python, on utilisera le type **deque** du module `collections`.

- ▶ La classe deque (double-ended queue) du module collections est une structure de données qui permet d'effectuer les opérations suivantes en temps (moyen) constant :
 - ▶ L'ajout d'un élément en fin de file avec la méthode append()
 - ▶ L'accès à et la suppression d'un élément en tête de file avec la méthode popleft()
- mais aussi
- ▶ L'ajout d'un élément en tête de file avec la méthode appendleft()
 - ▶ L'accès à et la suppression d'un élément en queue de file avec la méthode pop().
 - ▶ Par contre, l'insertion, la suppression, et même l'accès à un élément en milieu de file prend temps $\Theta(n)$.

collections.deque

```
from collections import deque

d = deque([1])
print(type(d)) # <class 'collections.deque'>

d.append(2)
d.append(3)
print(d) # deque([1, 2, 3])

d.appendleft(4)
d.appendleft(5)
print(d) # deque([5, 4, 1, 2, 3])

print(d.pop()) # 3
print(d.popleft()) # 5
print(d) # deque([4, 1, 2])

d = deque()
print(d) # deque([])
print(d.pop()) # IndexError: pop from an empty deque
```

Opération	Temps de parcours moyen
ajout en fin de file <code>d.append(x)</code>	$\Theta(1)$
accès et suppression en fin de file <code>d.pop()</code>	$\Theta(1)$
ajout en début de file <code>d.appendleft(x)</code>	$\Theta(1)$
accès et suppression en début de file <code>d.popleft()</code>	$\Theta(1)$

. <https://wiki.python.org/moin/TimeComplexity>

```
1  from collections import deque
2
3  def BFS(G,s):
4      '''
5      Entree: graphe G en dict de listes d'adj, s sommet
6      Parcourt G en largeur, affiche les sommets parcourus
7      '''
8      n = len(G)
9      a_parcourir = deque([s])
10     vu = [0 for u in range(n)]
11     vu[s] = 1
12
13     while a_parcourir:
14         sommet = a_parcourir.popleft()
15         for u in G[sommet]:
16             if not vu[u]:
17                 a_parcourir.append(u)
18                 vu[u] = 1
19         print(sommet)
```

- ▶ Chaque sommet atteignable depuis s est ajouté exactement une fois à `a_parcourir` (garanti par l'instruction `if` des lignes 16-18).
- ▶ La boucle `while` itère autant de fois qu'il y a de sommets ajoutés à `a_parcourir` au cours d'un parcours; le nombre d'itérations est égal au nombre de sommets atteignables depuis s (donc n au pire des cas). En particulier, l'instruction `dequeue` de la ligne 14 est donc exécutée n fois.
- ▶ A chaque itération de la boucle `while`, la boucle `for` itère autant de fois que le nombre de voisins du sommet sous considération. Le corps de la boucle `for` est donc exécuté au pire des cas m fois (pour un graphe dirigé) ou $2m$ fois (pour un graphe non dirigé) à travers toutes les itérations de la boucle `while`.
- ▶ Le temps de parcours total est donc $\Theta(n + m)$ (au pire des cas).

DFS - temps de parcours

```
def DFS(G, s):  
    print(s)  
    vu[s] = 1  
    for u in G[s]:  
        if not vu[u]:  
            DFS(G, u)
```

On peut prouver que le temps de parcours total de cet algorithme est également $\Theta(n + m)$ au pire des cas.

- ▶ On s'intéresse aux problèmes suivants : étant donné un graphe G représenté par des listes d'adjacence, et deux sommets u et v de G :
 - ▶ Donner un chemin entre u et v si un tel chemin existe.
 - ▶ Donner un plus court chemin entre u et v .
 - ▶ Donner la distance entre u et v .
- ▶ On va faire plus : étant donné G et un sommet s de G , on donne un plus court chemin entre s et v pour chaque sommet v de G atteignable depuis s .

- ▶ Pour trouver un plus court chemin entre deux sommets, on peut utiliser un algorithme qui s'inspire soit de BFS soit de DFS. Mais les propriétés de BFS garantissent que le **premier** chemin trouvé est le plus court.
- ▶ On définit un algorithme `BFS_chemin` qui prend en entrée un dictionnaire de listes d'adjacences d'un graphe G , et un sommet s de G , et retourne une liste chemin indexée par les sommets de G telle que pour tout sommet v
 - ▶ si v est atteignable depuis s , `chemin[v]` est une liste de sommets qui définissent un plus court chemin entre s et v ;
 - ▶ sinon, `chemin[v] = []`.
- ▶ Remarque : cet algorithme n'est pas le plus efficace pour calculer le plus court chemin.

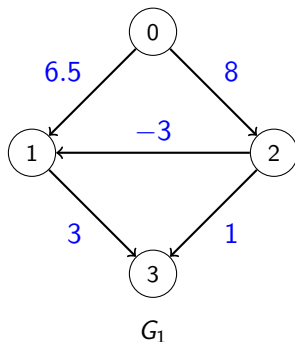
Plus court chemin

```
from collections import deque

def BFS_chemins(G,s):
    n = len(G)
    a_parcourir = deque([s])
    vu = [0 for u in range(n)]
    vu[s] = 1
    chemin = [[] for u in range(n)]
    chemin[s] = [s]
    while a_parcourir:
        sommet = a_parcourir.popleft()
        for u in G[sommet]:
            if not vu[u]:
                a_parcourir.append(u)
                vu[u] = 1
                chemin[u] = chemin[sommet].copy()
                chemin[u].append(u)
    return chemin
```

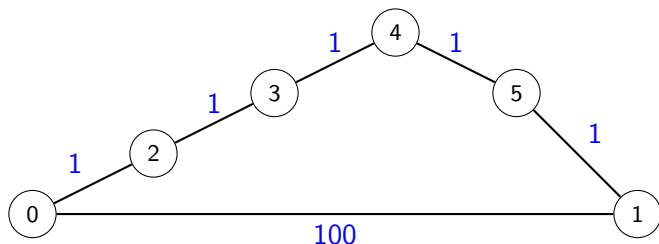
- ▶ Pour un graphe G pondéré, la **longueur** d'un chemin $u_0, u_1, \dots, u_\ell$ ($\ell \geq 1$) est la somme des poids des arêtes (u_i, u_{i+1}) .
- ▶ Pour u et v des sommets de G , un **plus court chemin** entre u et v est un chemin entre u et v de longueur minimale.
- ▶ La **distance** entre u et v est la longueur d'un plus court chemin entre u et v .

Exemple



- ▶ Il existe plusieurs chemins entre 0 et 3 :
 - ▶ 0, 1, 3 de longueur 9.5
 - ▶ 0, 2, 3 de longueur 9
 - ▶ 0, 2, 1, 3 de longueur 8.
- ▶ Le plus court chemin entre 0 et 3 est 0, 2, 1, 3.
- ▶ La distance entre 0 et 3 est 8.

Plus court chemin - graphes pondérés



- ▶ Dans un graphe non pondéré, un plus court chemin entre deux sommets u et v est un chemin avec le plus petit nombre possible d'arêtes, et on peut parcourir le graphe en largeur pour trouver un plus court chemin entre u et v .
- ▶ Dans un graphe pondéré, un parcours en largeur trouvera un chemin entre u et v avec le plus petit nombre possible d'arêtes mais pas nécessairement un plus court chemin !

Plus court chemin - graphes pondérés

Soit G un graphe dirigé et pondéré à n sommets et m arêtes, et s un sommet de G .

- ▶ Si tous les poids des arêtes sont positifs, **l'algorithme de Dijkstra**³ trouve le plus court chemin entre s et u pour tout u en temps $\mathcal{O}((n + m) \log_2(n))$.
- ▶ **L'algorithme de Bellman-Ford**⁴ permet les poids négatifs et calcule un plus court chemin de s à u pour tout u en temps $\mathcal{O}(nm)$.
- ▶ **L'algorithme de Floyd-Warshall**⁵ prend en entrée la matrice d'adjacence de G , permet les poids négatifs⁶ et calcule un plus court chemin de u à v pour **toutes les paires** de sommets u et v en temps $\Theta(n^3)$.

3. https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

4. https://fr.wikipedia.org/wiki/Algorithme_de_Bellman-Ford

5. https://fr.wikipedia.org/wiki/Algorithme_de_Floyd-Warshall

6. mais pas les cycles de poids négatif

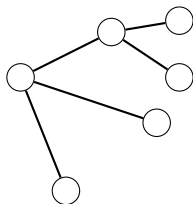
Cycles, graphes acycliques

- Dans un graphe, un **cycle** est un chemin d'un sommet u à lui-même, c'est-à-dire une suite de sommets

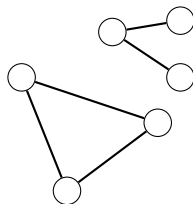
$$u_0, u_1, \dots, u_\ell, \quad \ell \geq 1$$

tels que $u_0 = u_\ell = u$, les u_i pour $0 < i < \ell$ sont tous distincts, et u_i et u_{i+1} sont liés par une arête pour tout $0 \leq i \leq \ell - 1$.

- Un graphe sans cycles est dit **acyclique**.

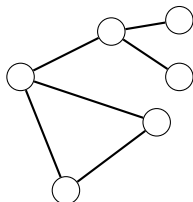


Un graphe acyclique

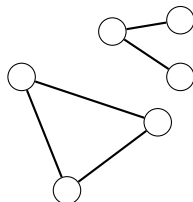


Un graphe avec cycle

- ▶ Dans ce qui suit, soit G un graphe non dirigé.
- ▶ G est dit **connexe** si pour toute paire de sommets u, v , il existe un chemin entre u et v .

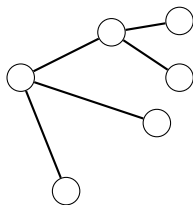


Un graphe connexe

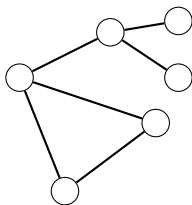


Un graphe non connexe

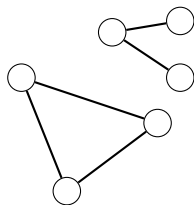
- ▶ Soit G un graphe non dirigé. Si G est connexe et acyclique, G est dit un **arbre**.
- ▶ Parmi les graphes ci-dessous, seul G_1 est un arbre. G_2 est connexe mais contient un cycle, et G_3 est non connexe (et contient un cycle).



G_1



G_2

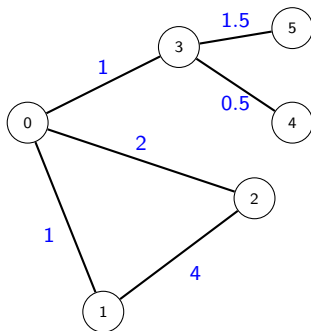


G_3

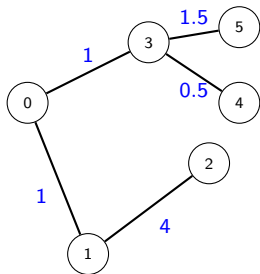
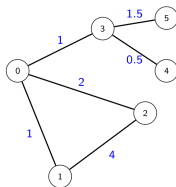
- ▶ Soit G un graphe pondéré, connexe. Un **arbre couvrant** de G est un graphe T qui a
 - ▶ les mêmes sommets que G
 - ▶ et un sous-ensemble des arêtes de G ,tel que T est un arbre.
- ▶ **Un arbre couvrant minimal** (ou arbre couvrant de poids minimal) de G est un arbre couvrant T tel que la somme des poids des arêtes de T est minimale.

Arbre couvrant minimal - exemple

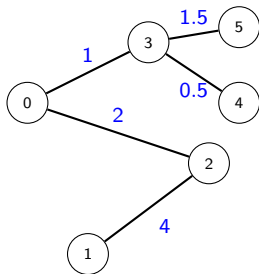
Donner un arbre couvrant minimal du graphe ci-dessous :



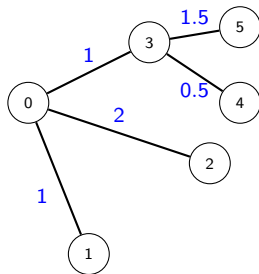
Arbre couvrant minimal



Un arbre couvrant de poids 8



Un arbre couvrant de poids 9



Un arbre couvrant **minimal** de poids 6

- ▶ Pour un graphe G non dirigé et pondéré, le problème de trouver un arbre couvrant minimal de G a un grand nombre d'applications⁷.
- ▶ il existe des algorithmes⁸ qui trouvent un arbre couvrant minimal d'un graphe à n sommets et m arêtes en temps $\mathcal{O}(m \log_2(n))$.

7. https://en.wikipedia.org/wiki/Minimum_spanning_tree#Applications

8. https://fr.wikipedia.org/wiki/Algorithme_de_Prim,
https://fr.wikipedia.org/wiki/Algorithme_de_Kruskal

Arbre couvrant minimal avec BFS/DFS

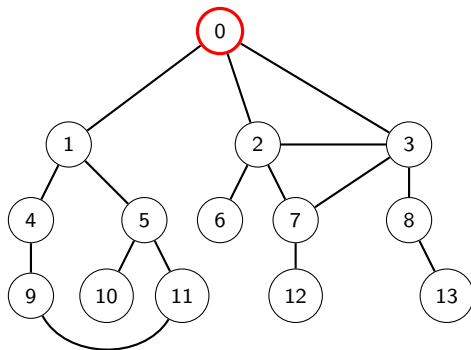
- ▶ Soit G un graphe connexe. Si G est non pondéré, tous les arbres couvrants sont de même poids. En particulier, tout arbre couvrant est minimal.
- ▶ Dans ce cas, BFS et DFS produisent un arbre couvrant minimal, qui reflète dans chaque cas le parcours de chaque algorithme.

Arbre couvrant avec BFS

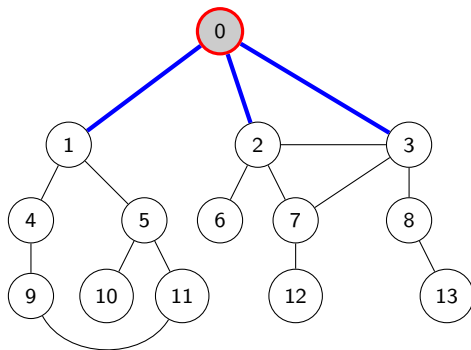
```
from collections import deque

def BFS_arbre(G,s):
    '''Entree: G non dirige en listes d'adj, s sommet
    Sortie: arbre couvrant T'''
    n = len(G)
    a_parcourir = deque([s])
    vu = [0 for u in range(n)]
    vu[s] = 1
    T = {u:[] for u in range(n)}
    while a_parcourir:
        sommet = a_parcourir.popleft()
        for u in G[sommet]:
            if not vu[u]:
                a_parcourir.append(u)
                vu[u] = 1
                T[sommet].append(u)
                T[u].append(sommet)

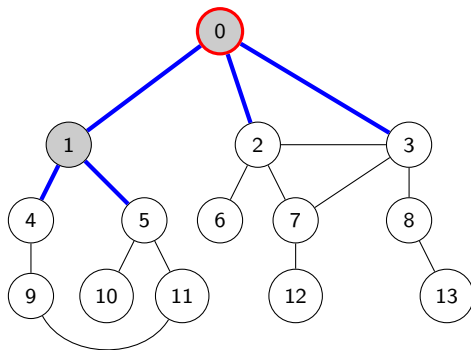
    return T
```

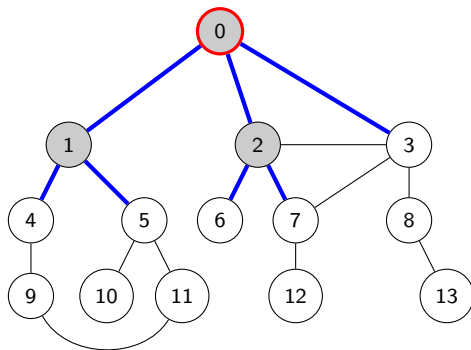
`a_parcourir = [0]`



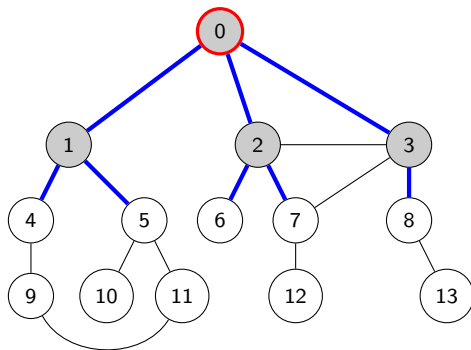
a_parcourir = [1, 2, 3]



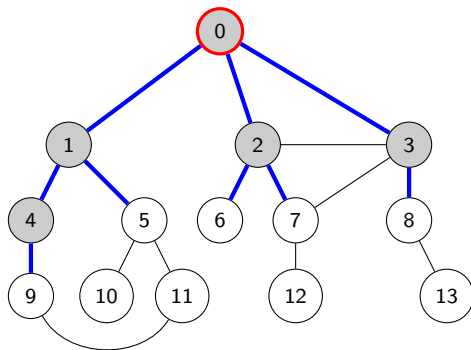
a_parcourir = [2, 3, 4, 5]



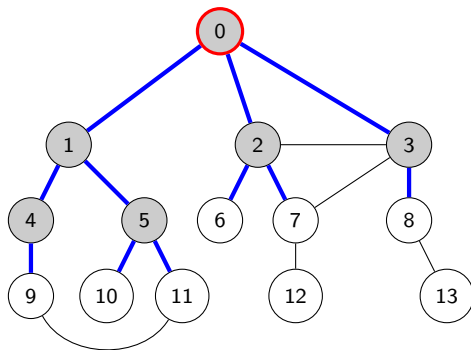
a_parcourir = [3, 4, 5, 6, 7]



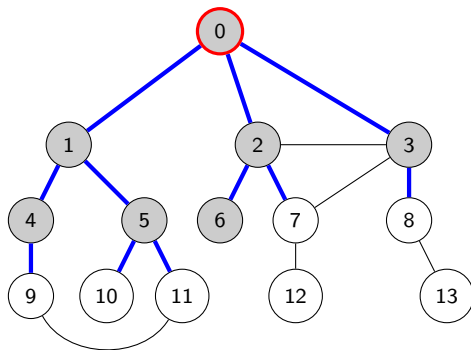
a_parcourir = [4, 5, 6, 7, 8]



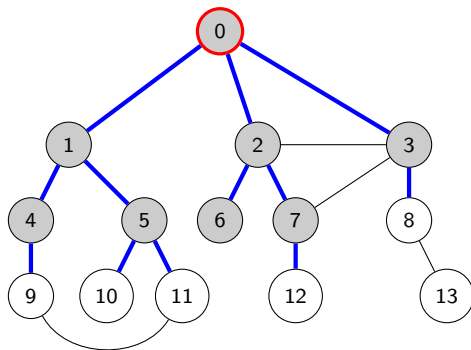
a_parcourir = [5, 6, 7, 8, 9]



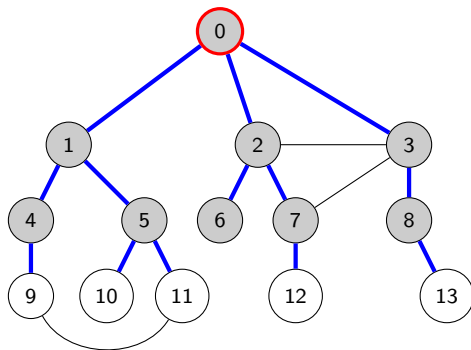
a_parcourir = [6, 7, 8, 9, 10, 11]



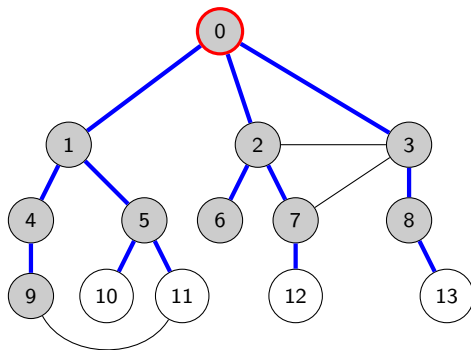
`a_parcourir = [7, 8, 9, 10, 11]`



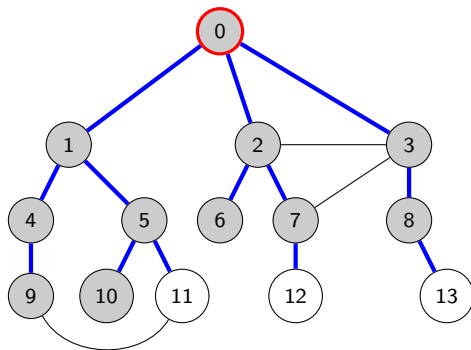
`a_parcourir = [8, 9, 10, 11, 12]`



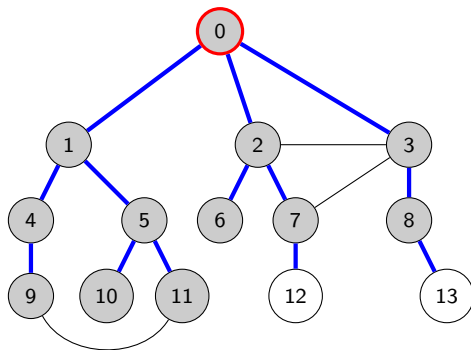
`a_parcourir = [9, 10, 11, 12, 13]`



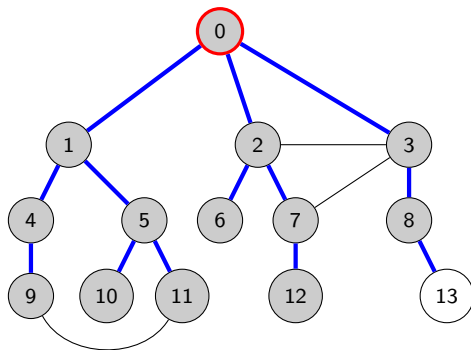
a_parcourir = [**10**, 11, 12, 13]



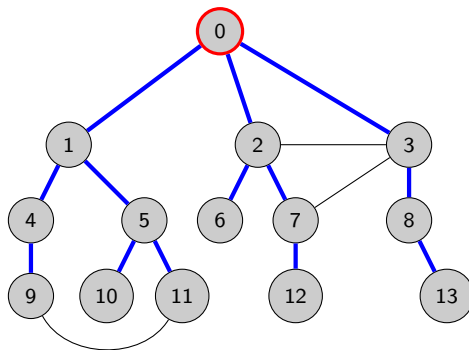
a_parcourir = [11, 12, 13]



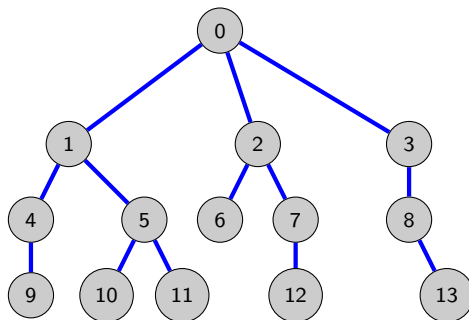
a_parcourir = [12, 13]



`a_parcourir = [13]`

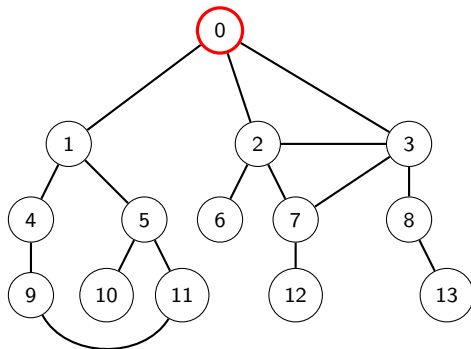


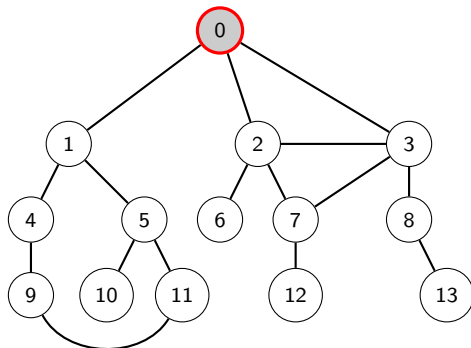
```
a_parcourir = []
```

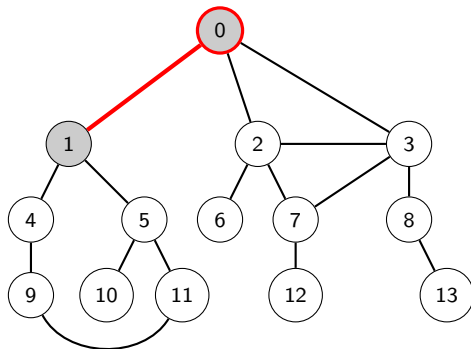


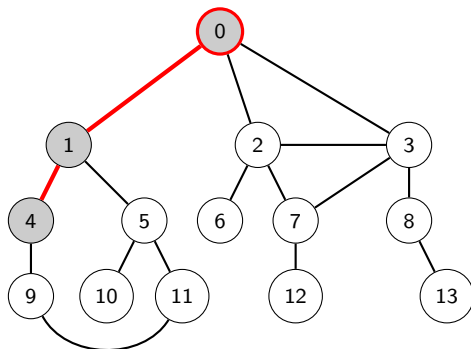
Arbre couvrant avec DFS

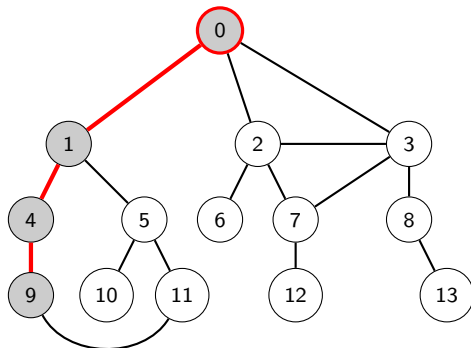
```
def DFS_arbre(G, s):  
    '''  
    Entree: graphe G non dirige, s sommet  
    Sortie: arbre couvrant T  
    '''  
  
    vu[s] = 1  
    for u in G[s]:  
        if not vu[u]:  
            T[s].append(u)  
            T[u].append(s)  
            DFS_arbre(G, u)  
  
G = {0:[1,2], 1:[0,2], 2:[0, 1]}  
vu = [0 for u in G]  
T = {u:[] for u in G}  
DFS_arbre(G, 0)  
print(T)
```

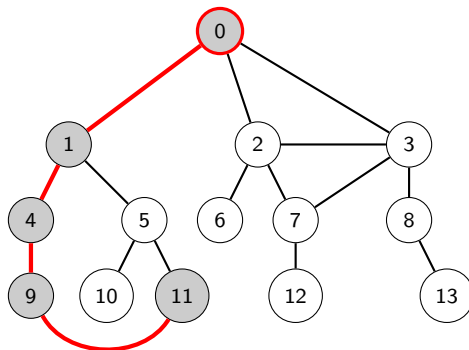


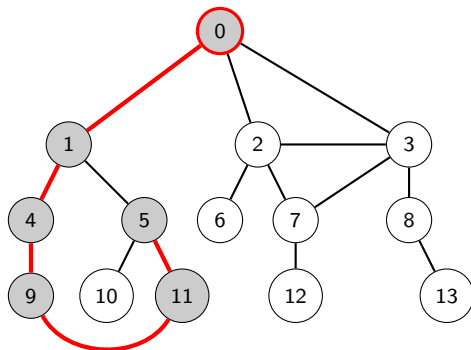


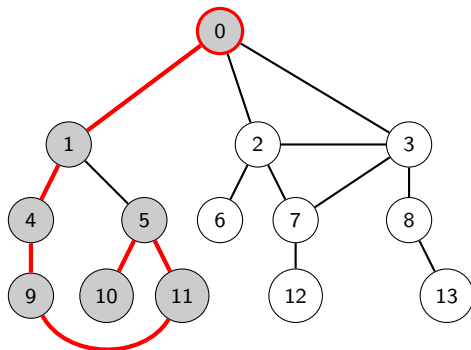


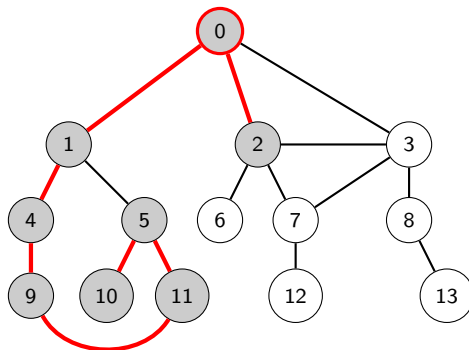


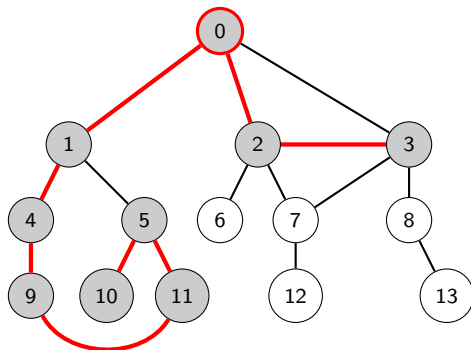


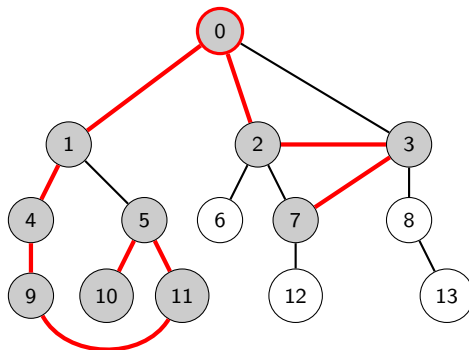


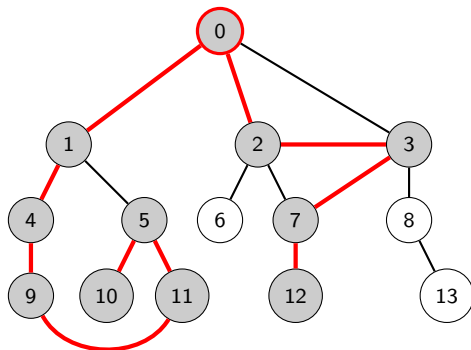


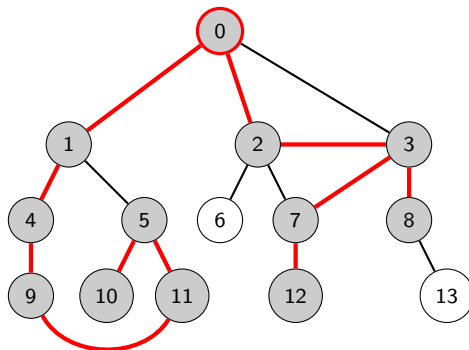


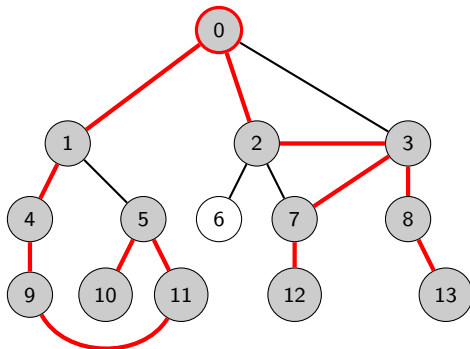


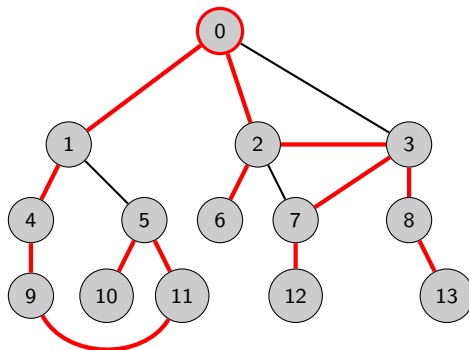


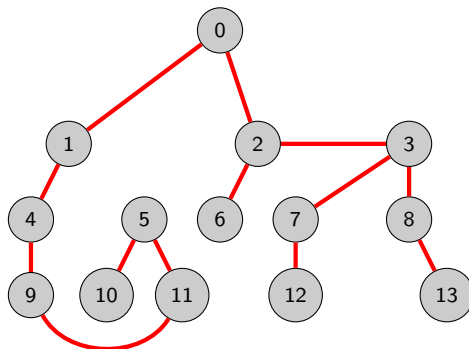




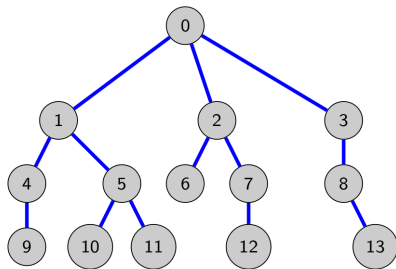




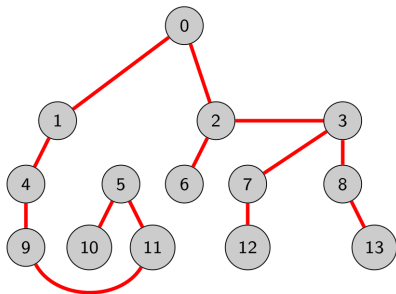




Arbres couvrants



Obtenu avec un parcours en largeur



Obtenu avec un parcours en
profondeur