Programmation d'une Intelligence Artificielle : arbres ou apprentissage automatique

Graphes : vocabulaire et parcours.

Paul Mangold

L3 MIASHS

29 Janvier 2021

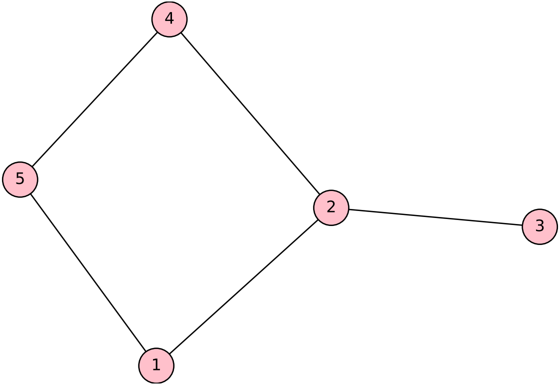
Graphes : défintions

Formellement, un graphe c'est $G = (V, E)$, où

- V est l'ensemble des **sommets** de G ;
- E est l'ensemble des **arêtes** de G.

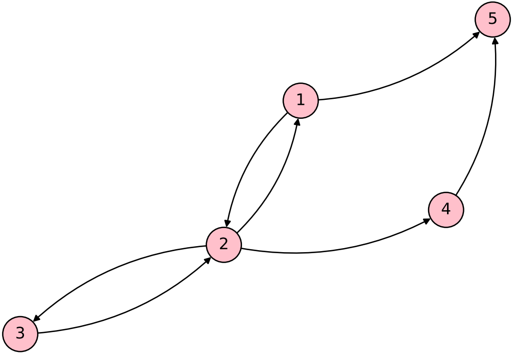
Les arêtes peuvent être **non orientées** :

- $V = \{1,\ 2,\ 3,\ 4,\ 5\}$;
- $E = \{\{1, 2\},\ \{2, 3\},\ \{2, 4\},\ \{4, 5\},\ \{1, 5\}\}$.

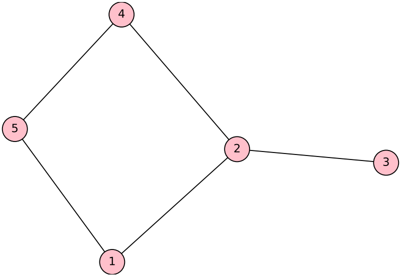


Ou elles peuvent être **orientées** :

- $V = \{1,\ 2,\ 3,\ 4,\ 5\}$;
- $E = \{(1,2),\ (2,1),\ (2,3),\ (3,2),\ (2,4),\ (4,5),\ (1,5)\}$.

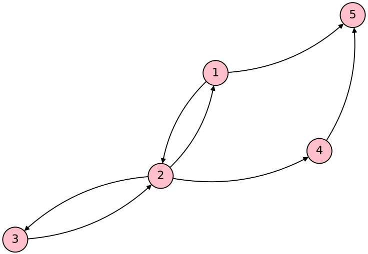


Dans un graphe $G = (V, E)$, le **degré** d'un sommet $s \in V$ est le nombre d'arêtes reliant ce sommet. On le note $deg(s)$.



Par exemple : $deg(2) = 3$.

Si le graphe est orienté, c'est la même chose :

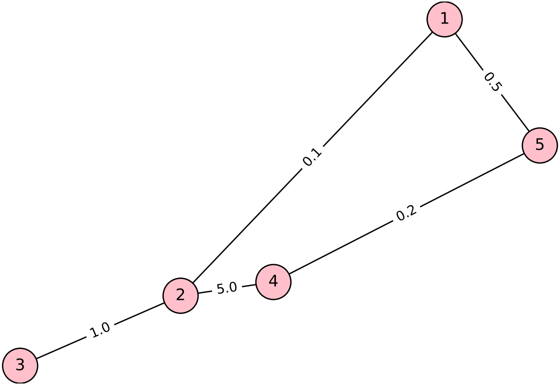


Ici, $deg(2) = 5$.

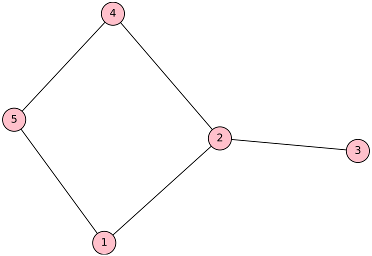
Graphe pondéré

Un graphe **pondéré** est un graphe $G = (V, E, w)$ où

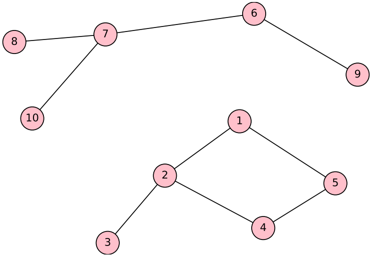
- $w : E \to \mathbb{R}$ est une fonction à valeurs réelles ;
- Pour $(u, v) \in E$, $w(u, v)$ est le poids de (u, v).



S'il existe un chemin entre toutes les paires de sommets, le graphe est **connexe**.
Pour un graphe orienté, on oublie l'orientation des arêtes. Mais si même en gardant l'orientation, les chemins existent, il est **fortement connexe**.

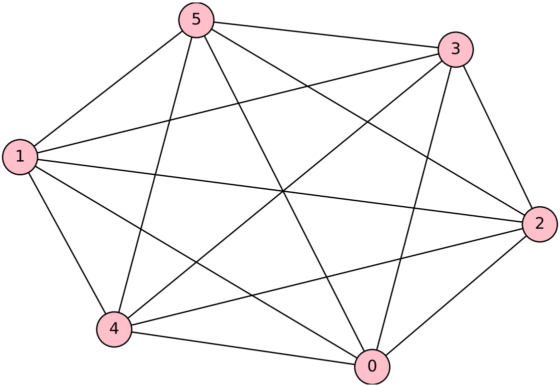


(a) Connexe

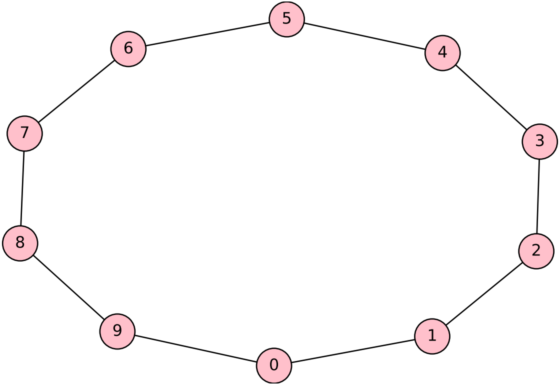


(b) Non Connexe

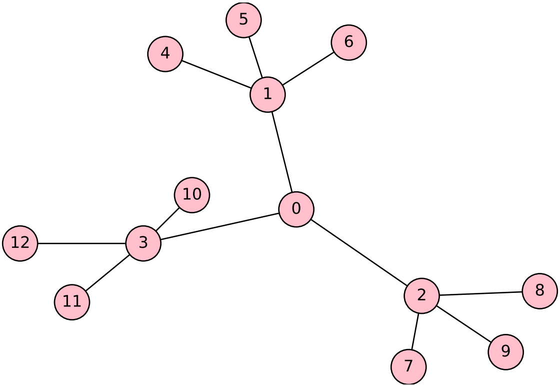
S'il y a des arêtes entre tous les sommets, le graphe est **complet**.



S'il y a autant d'arêtes que de sommets, et que chaque sommet est de degré 2, le graphe est un **cycle**.



Un **arbre** est un graphe orienté, acyclique et connexe.



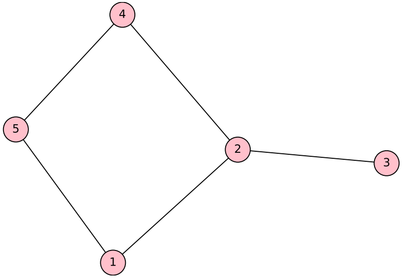
Représentation d’un Graphe

Un graphe est un ensemble de sommets et d'arêtes : $G = (V, E)$.

Problème : pour trouver les sommets reliés à un sommet s, on doit parcourir tout l'ensemble E... ce n'est pas très efficace !

Gardons plutôt en mémoire les sommets auxquels sont reliés chacun des sommets. Pour un sommet $s \in V$:

$$adj(s) = \{s' \in V \mid (s, s') \in E\}$$



Ici, $adj(2) = \{1, 3, 4\}$.

Ou alors, on peut tout stocker sous la forme d'une matrice :

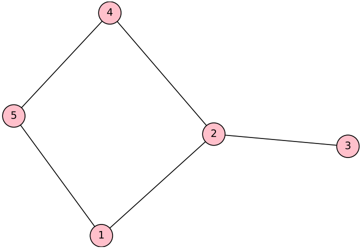


Figure: Graphe

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure: Matrice d'Adjacence

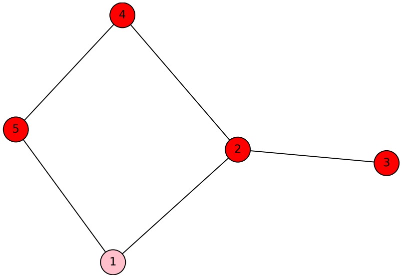
Chemins

Chemins

Chemin

Un chemin entre deux sommets $s, s' \in V$ est une suite de sommets $(s_1, \ldots, s_k)$ tels que

- $s_1 = s$;
- $s_k = s'$;
- pour tout $i \leq k - 1$, $(s_i, s_{i+1}) \in E$.



(3, 2, 4, 5) est un chemin de 3 à 5.

On n'a pas trop le choix... il faut se promener dans le graphe !

Parcours de Graphes

L'idée est d'explorer les voisins en se rappelant de

- quels sommets on a visités : ceux dont on a vu les voisins ;
- quels sommets on veut visiter bientôt : ceux qui sont voisins d'un sommet visité.

On garde les deux en mémoire dans des ensembles `visité` et `a_visiter`.

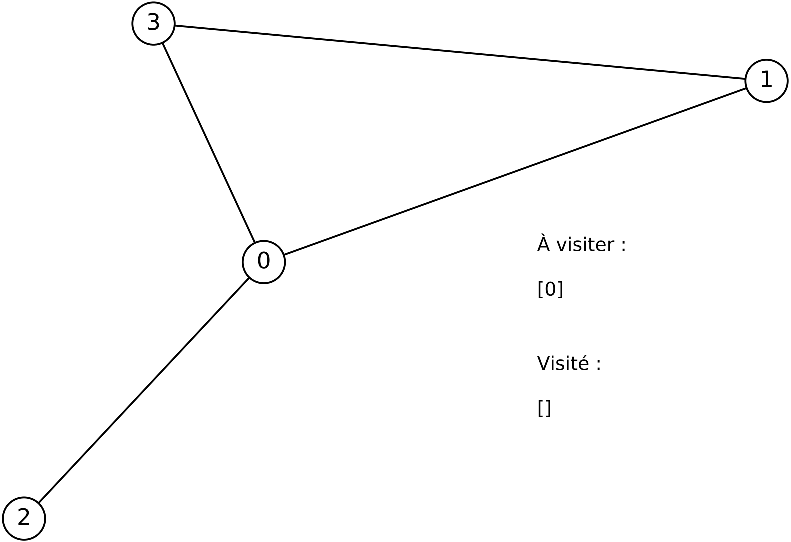
Pour parcourir depuis le sommet s :

- `visité` $= []$;
- `a_visiter` $= [s]$.

Tant que `a_visiter` n'est pas vide :

- prendre s dans `a_visiter` ;
- retirer s de `a_visiter` ;
- rajouter s dans `visité` ;
- rajouter les voisins $adj(s)$ **pas encore visités** dans `a_visiter`.

Quelle représentation de graphe utiliser pour ce type d'algorithme ?

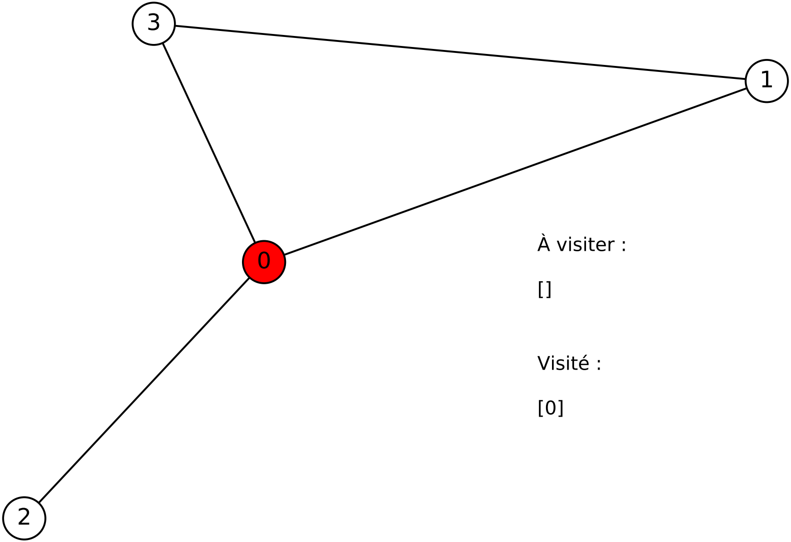


À visiter :

[0]

Visité :

[]



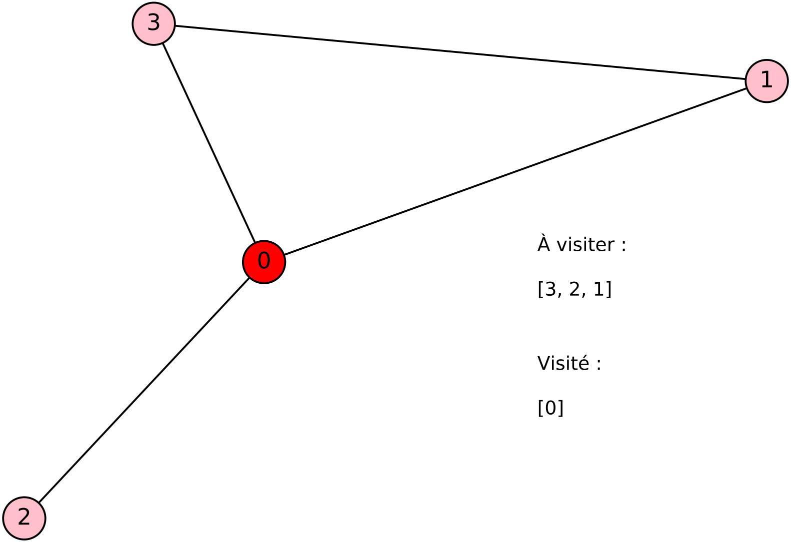
À visiter :

[]

Visité :

[0]

Exemple

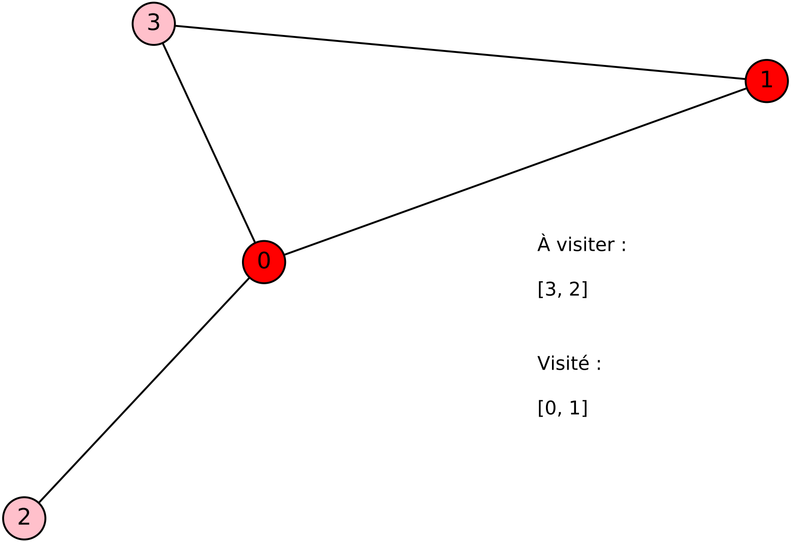


À visiter :

[3, 2, 1]

Visité :

[0]

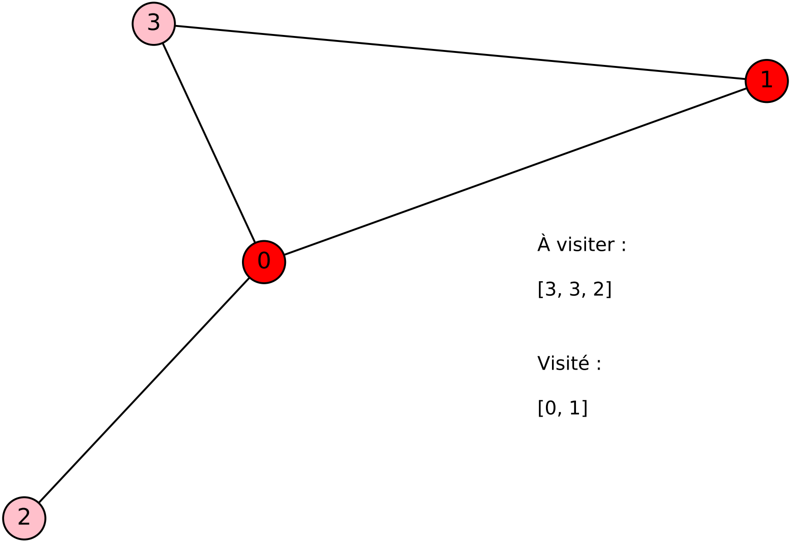


À visiter :

[3, 2]

Visité :

[0, 1]

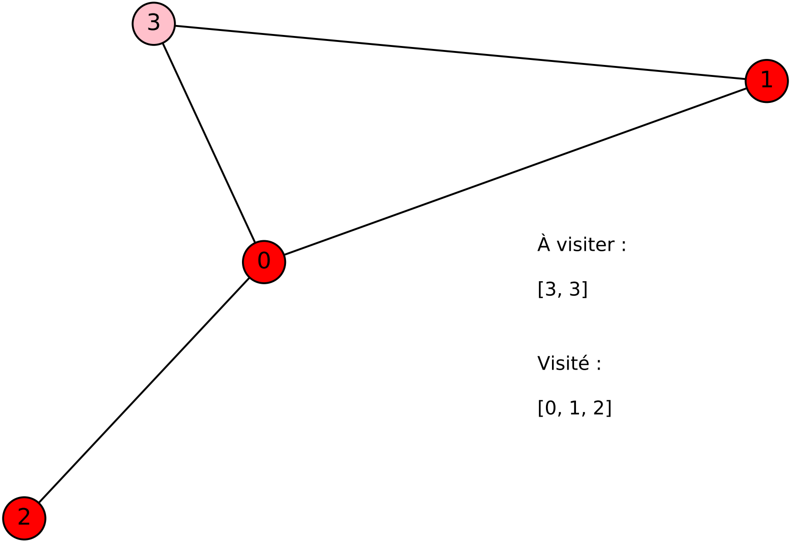


À visiter :

[3, 3, 2]

Visité :

[0, 1]

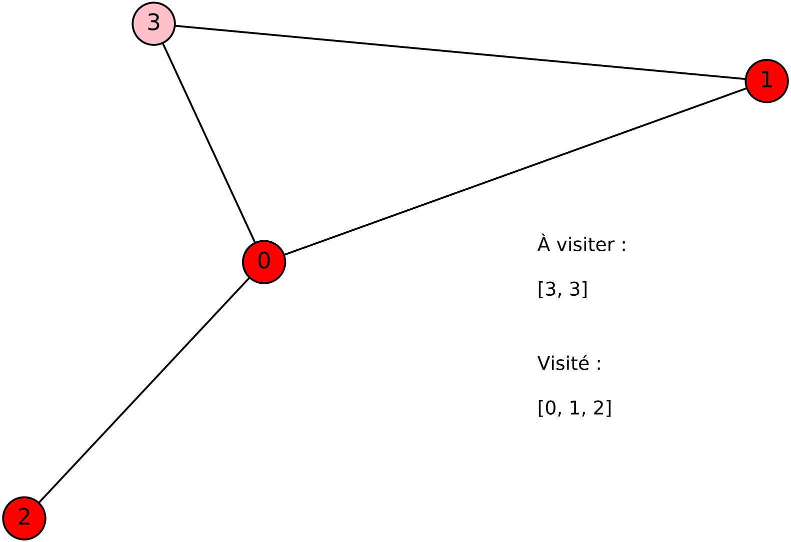


À visiter :

[3, 3]

Visité :

[0, 1, 2]

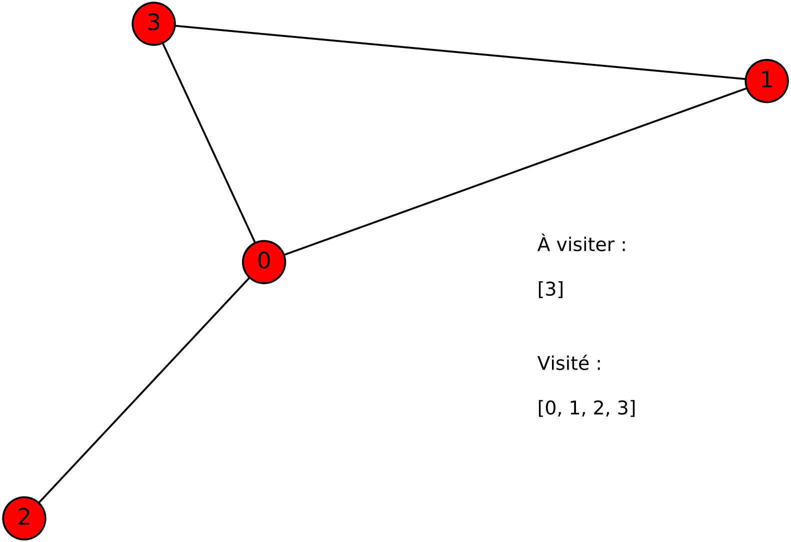


À visiter :

[3, 3]

Visité :

[0, 1, 2]

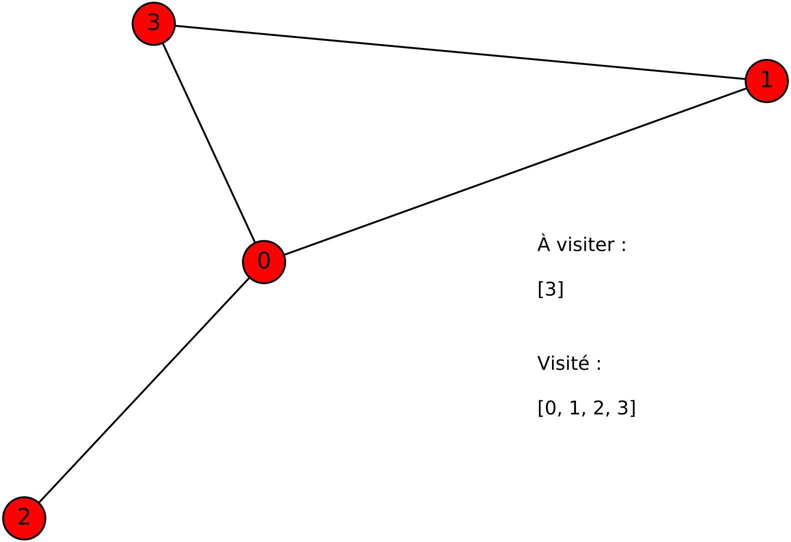


À visiter :

[3]

Visité :

[0, 1, 2, 3]

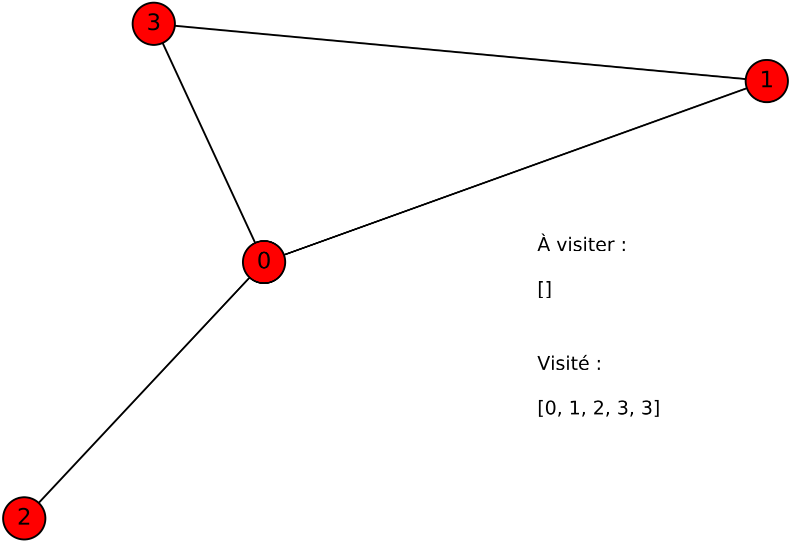


À visiter :

[3]

Visité :

[0, 1, 2, 3]



À visiter :

[]

Visité :

[0, 1, 2, 3, 3]

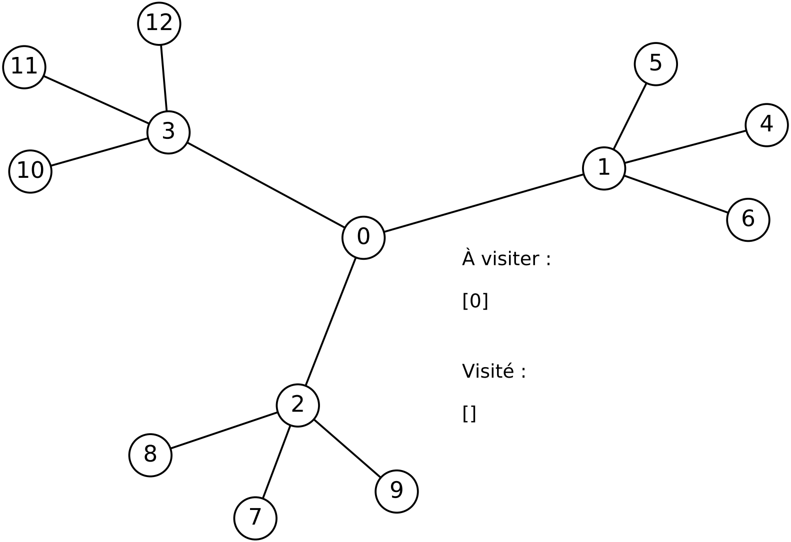
Dans un parcours en largeur, on parcourt d'abord les sommets les plus proches.

Tant que `a_visiter` n'est pas vide :

- $s =$ dernier élément de `a_visiter` ;
- retirer s de `a_visiter` ;
- rajouter s dans `visité` ;
- rajouter les voisins $adj(s)$ **au début de** `a_visiter`.

a_visiter fonctionne en "first in, first out" (FIFO) : les premiers éléments ajoutés sont les premiers à être explorés.

On explore tous les sommets les plus proches avant d'aller plus loin.

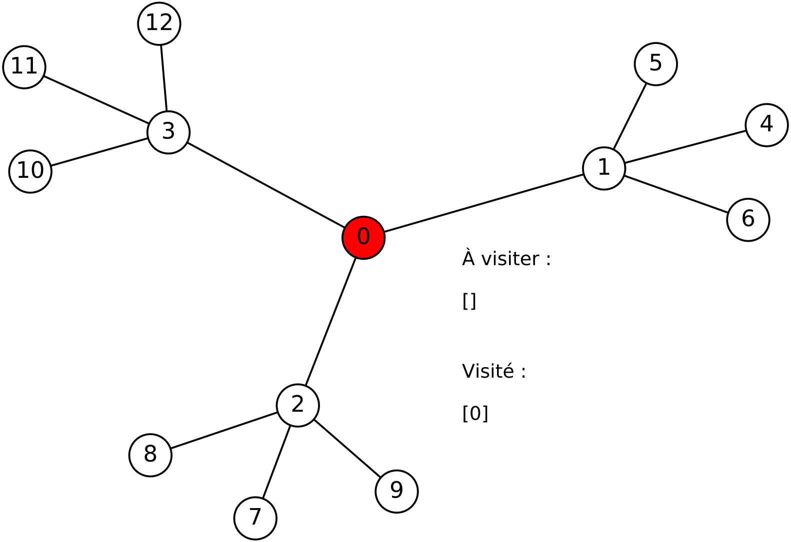


À visiter :

[0]

Visité :

[]

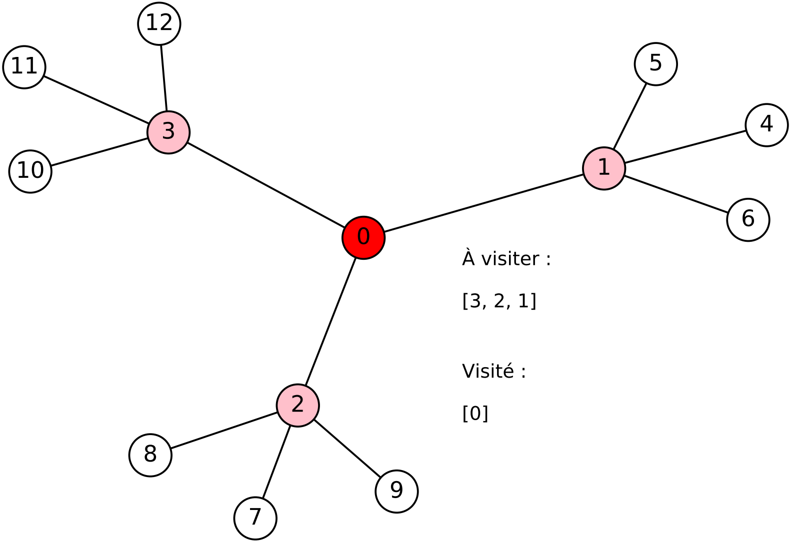


À visiter :

[]

Visité :

[0]

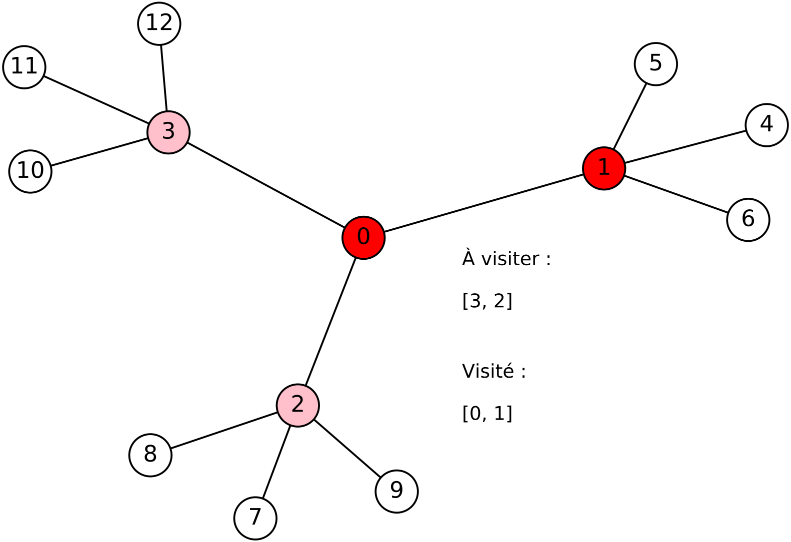


À visiter :

[3, 2, 1]

Visité :

[0]

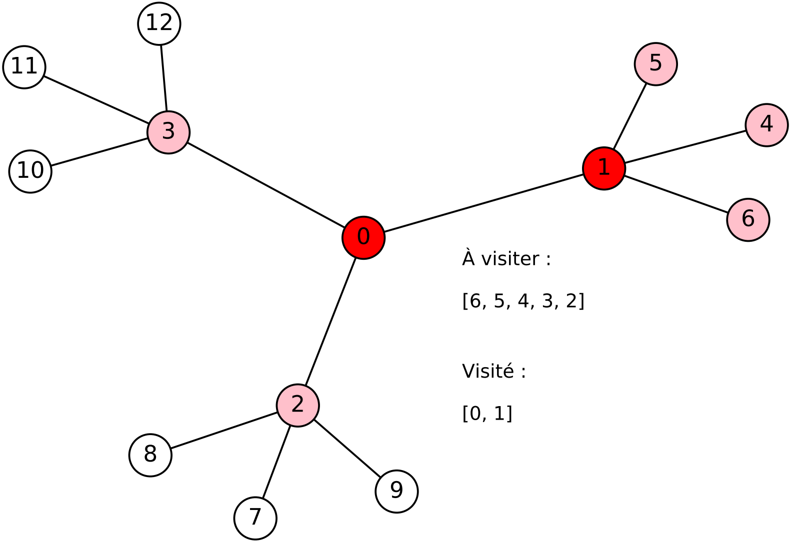


À visiter :

[3, 2]

Visité :

[0, 1]

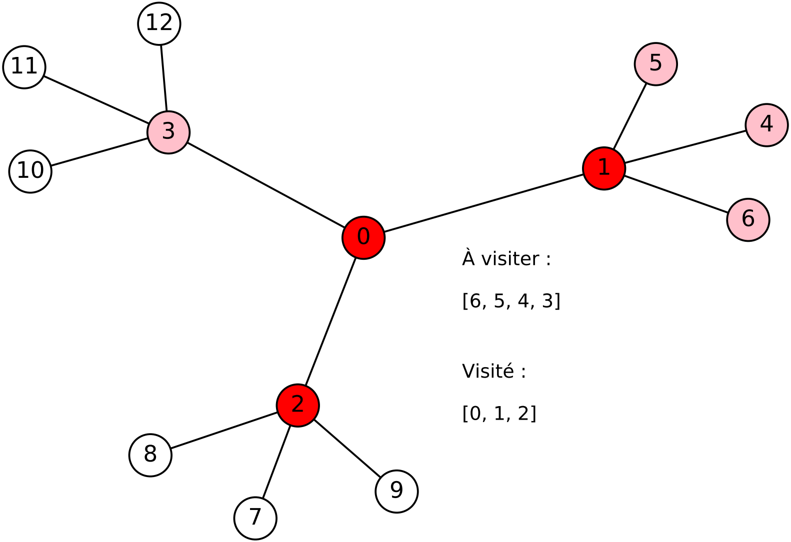


À visiter :

[6, 5, 4, 3, 2]

Visité :

[0, 1]

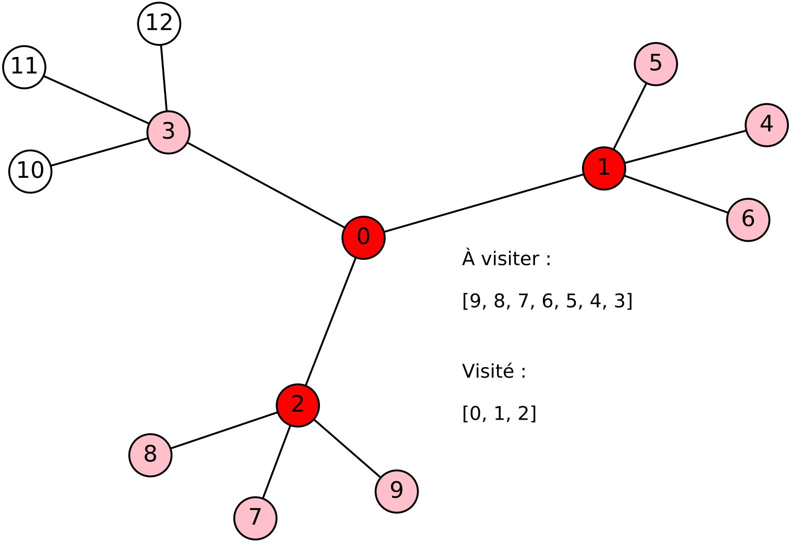


À visiter :

[6, 5, 4, 3]

Visité :

[0, 1, 2]

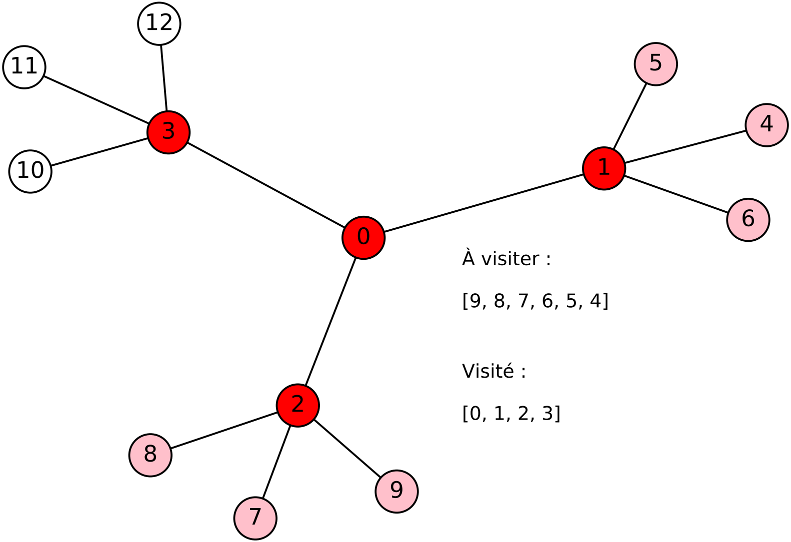


À visiter :

[9, 8, 7, 6, 5, 4, 3]

Visité :

[0, 1, 2]

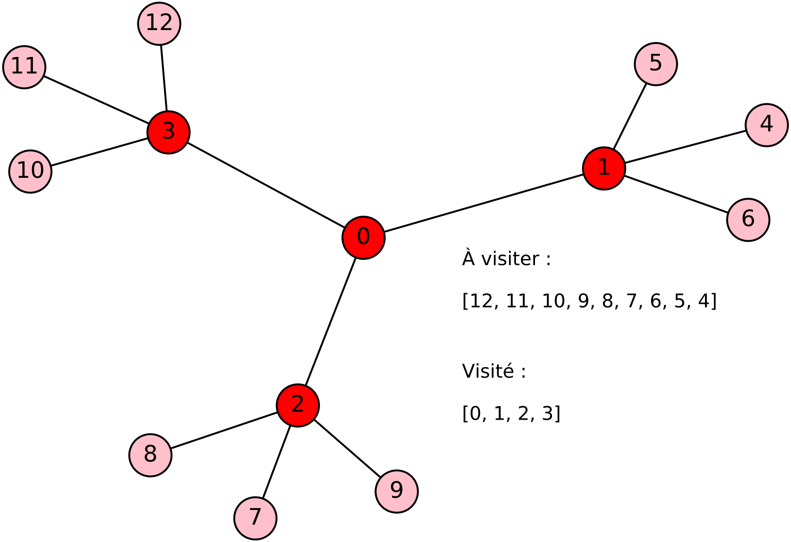


À visiter :

[9, 8, 7, 6, 5, 4]

Visité :

[0, 1, 2, 3]

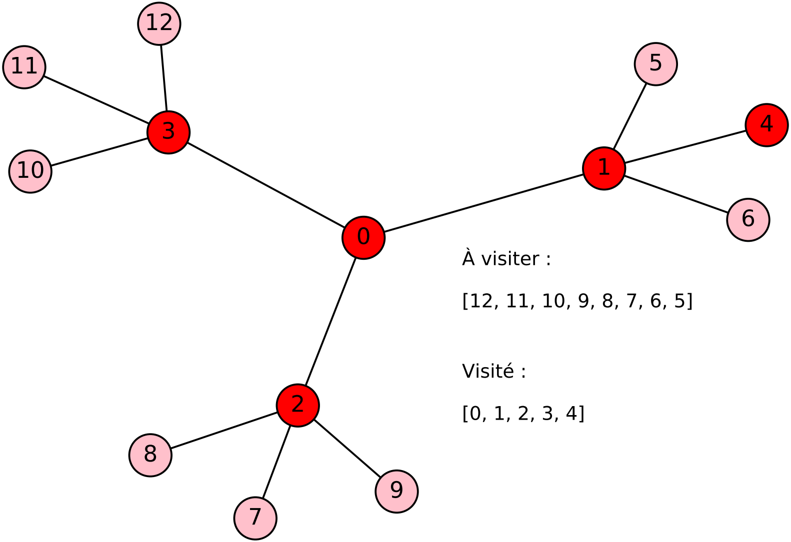


À visiter :

[12, 11, 10, 9, 8, 7, 6, 5, 4]

Visité :

[0, 1, 2, 3]

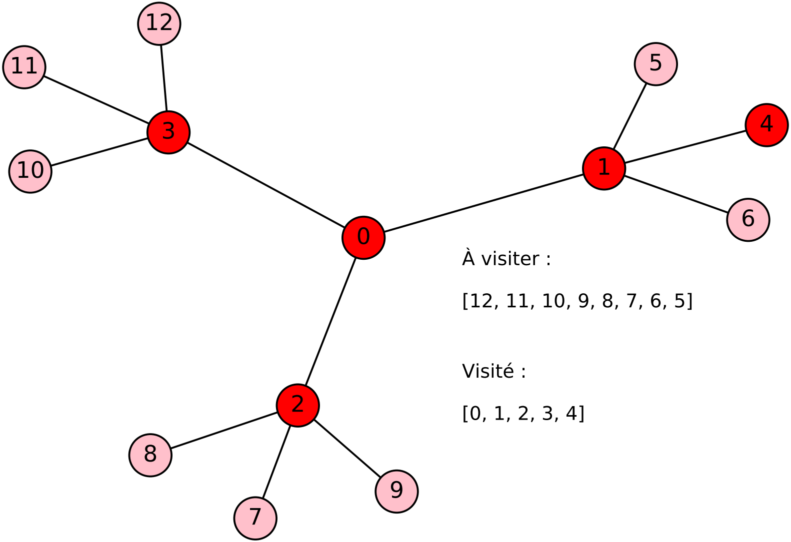


À visiter :

[12, 11, 10, 9, 8, 7, 6, 5]

Visité :

[0, 1, 2, 3, 4]

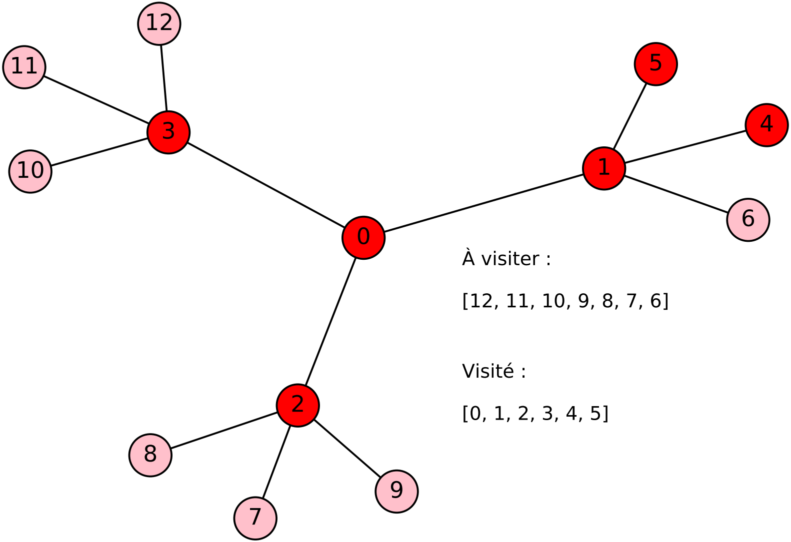


À visiter :

[12, 11, 10, 9, 8, 7, 6, 5]

Visité :

[0, 1, 2, 3, 4]

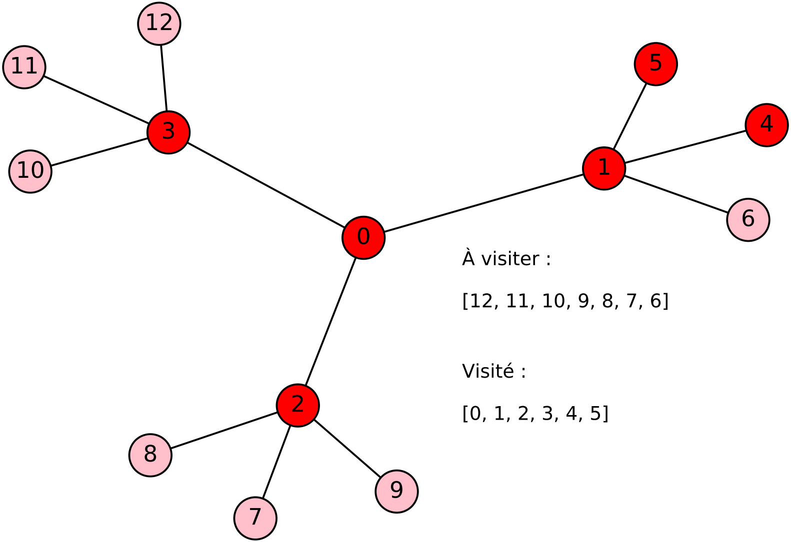


À visiter :

[12, 11, 10, 9, 8, 7, 6]

Visité :

[0, 1, 2, 3, 4, 5]

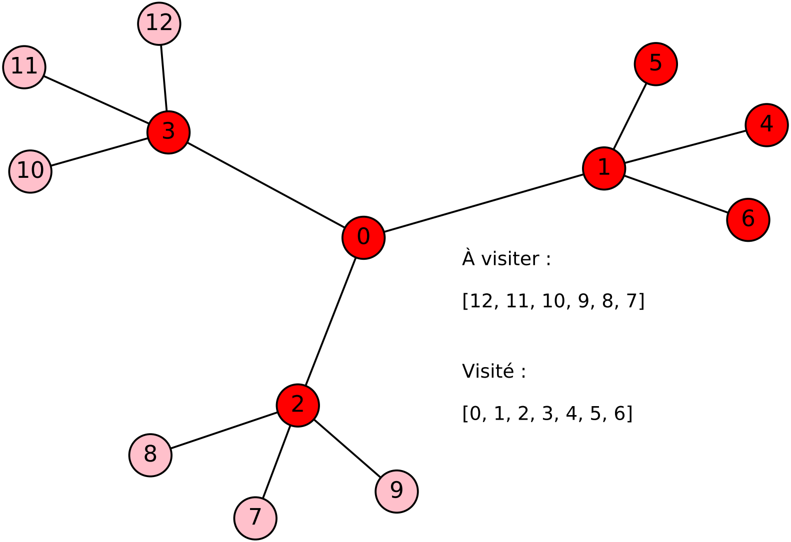


À visiter :

[12, 11, 10, 9, 8, 7, 6]

Visité :

[0, 1, 2, 3, 4, 5]

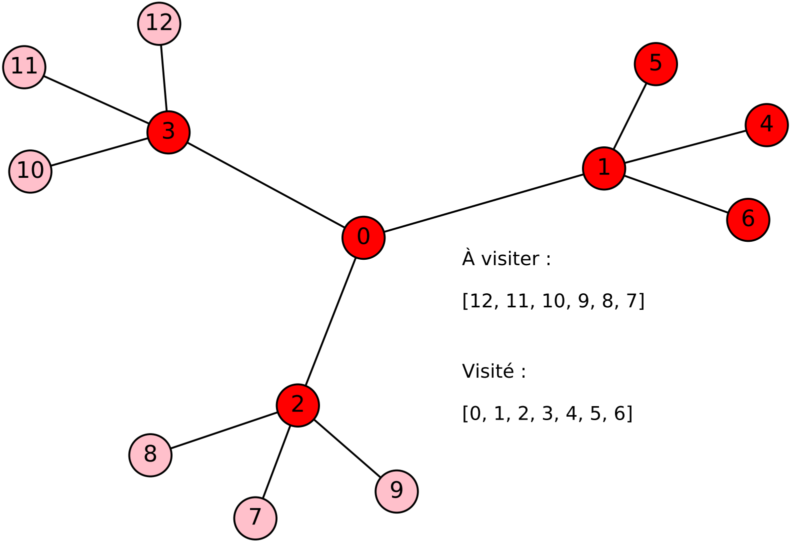


À visiter :

[12, 11, 10, 9, 8, 7]

Visité :

[0, 1, 2, 3, 4, 5, 6]

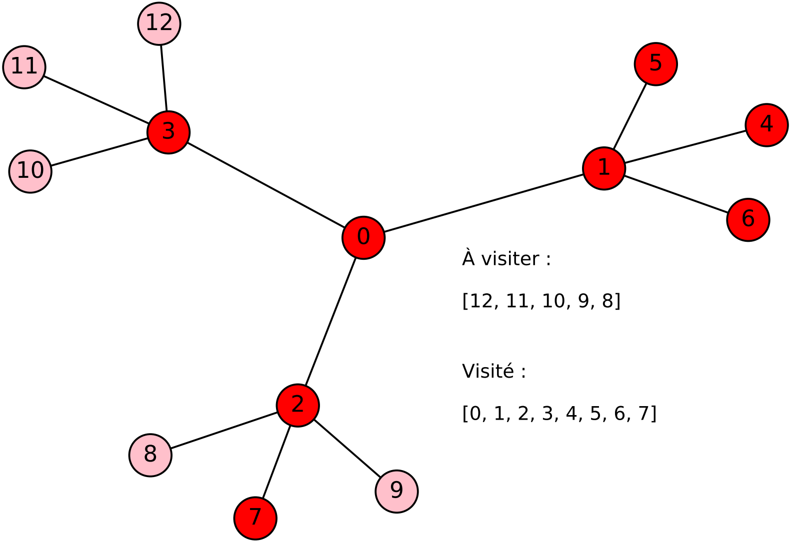


À visiter :

[12, 11, 10, 9, 8, 7]

Visité :

[0, 1, 2, 3, 4, 5, 6]

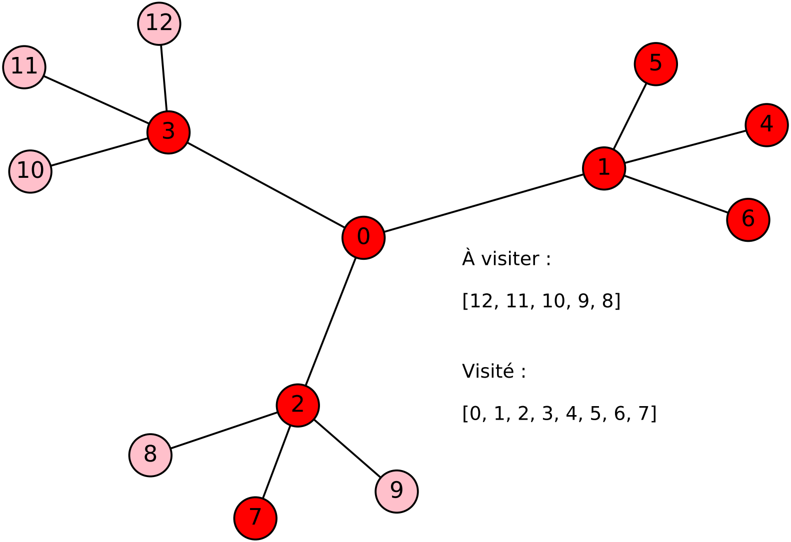


À visiter :

[12, 11, 10, 9, 8]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7]

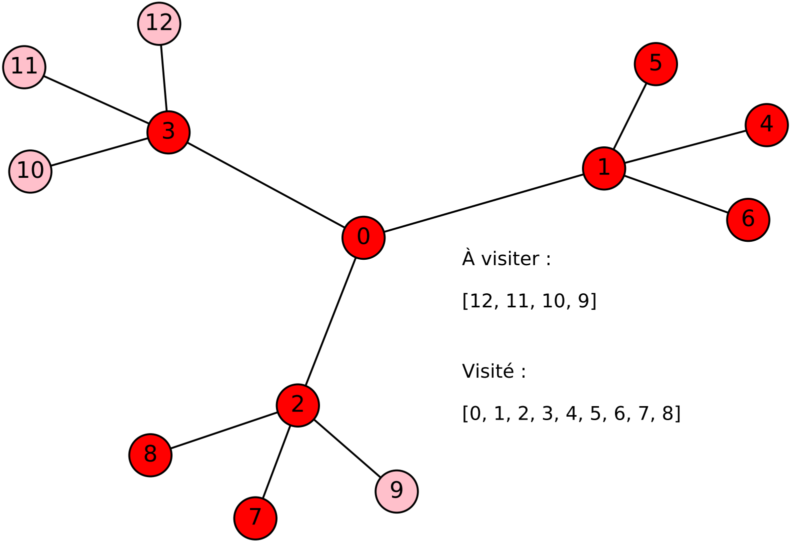


À visiter :

[12, 11, 10, 9, 8]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7]

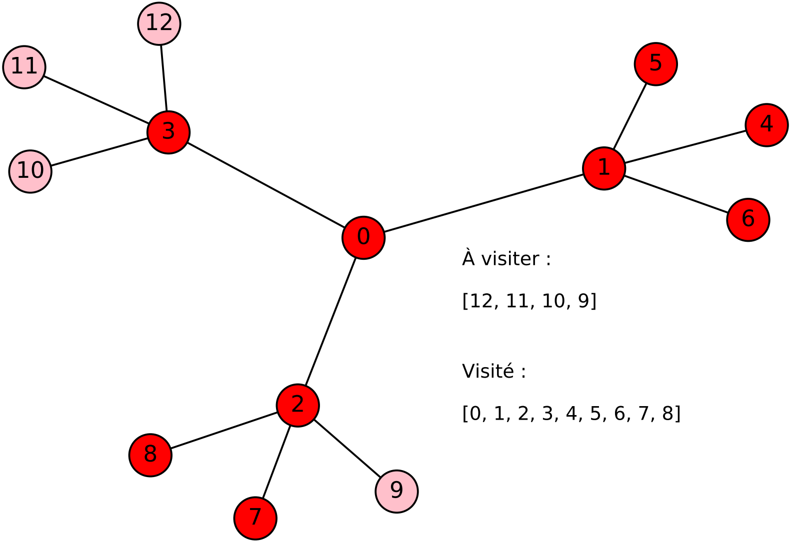


À visiter :

[12, 11, 10, 9]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7, 8]

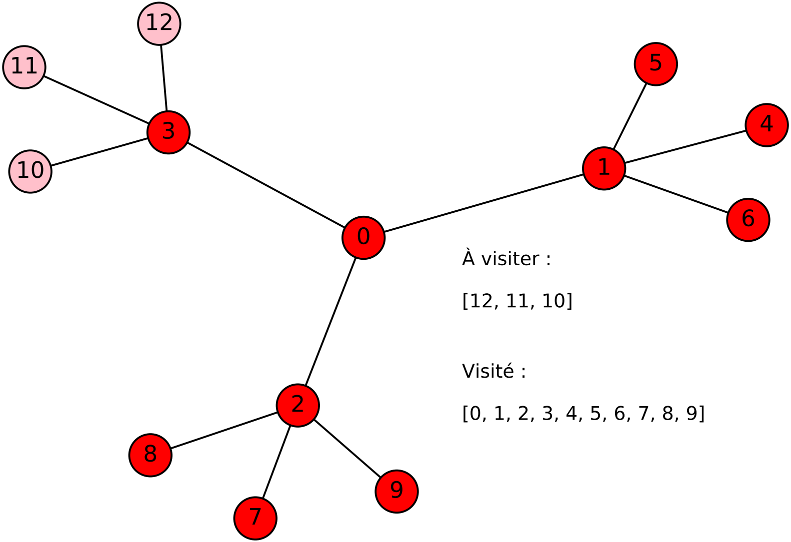


À visiter :

[12, 11, 10, 9]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7, 8]

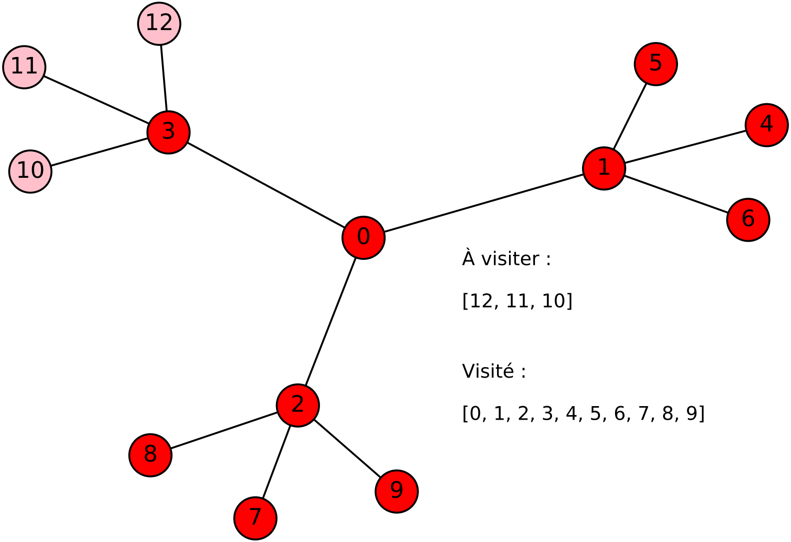


À visiter :

[12, 11, 10]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

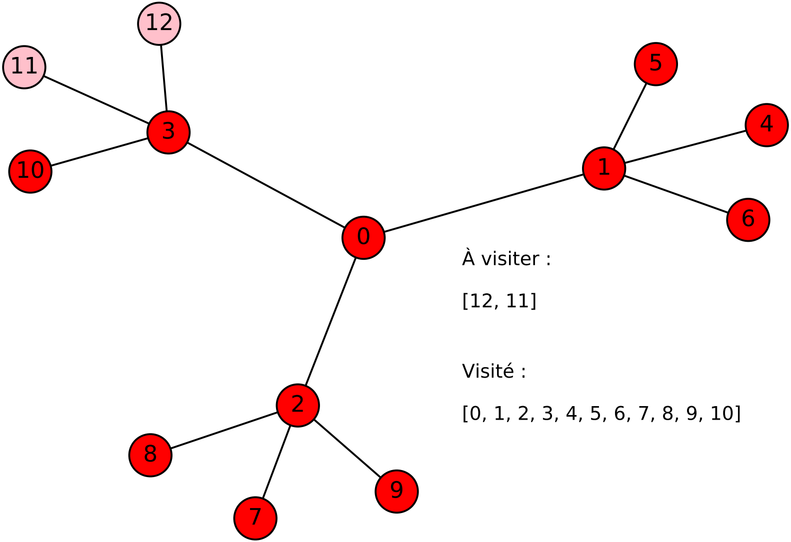


À visiter :

[12, 11, 10]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

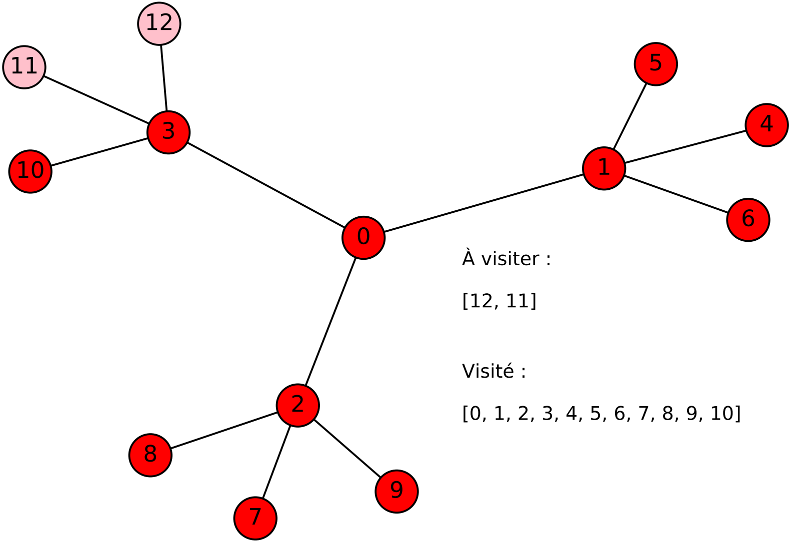


À visiter :

[12, 11]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

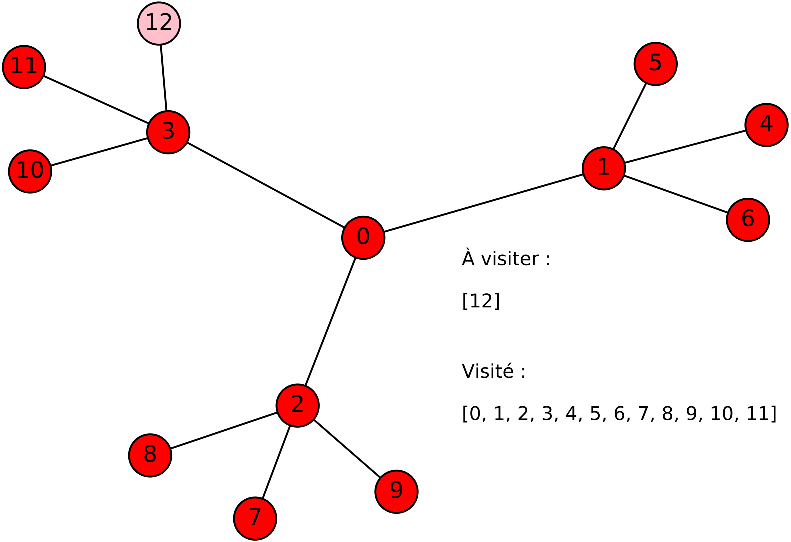


À visiter :

[12, 11]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

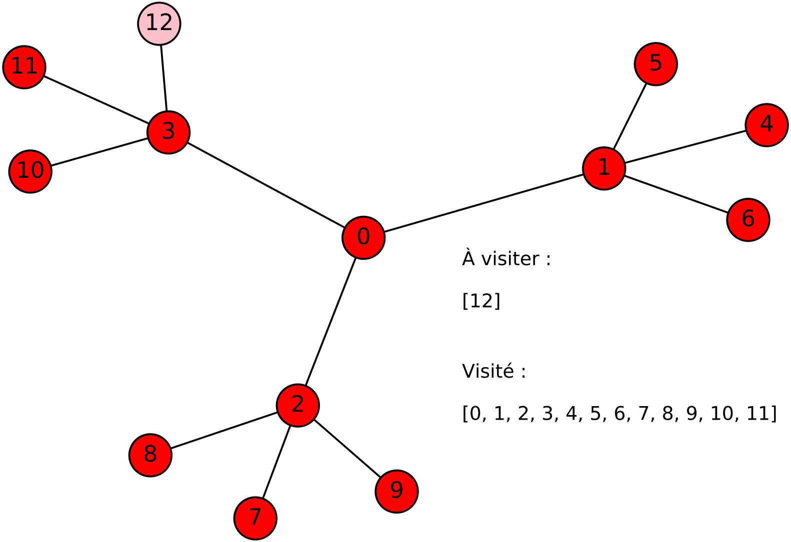


À visiter :

[12]

Visité :

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

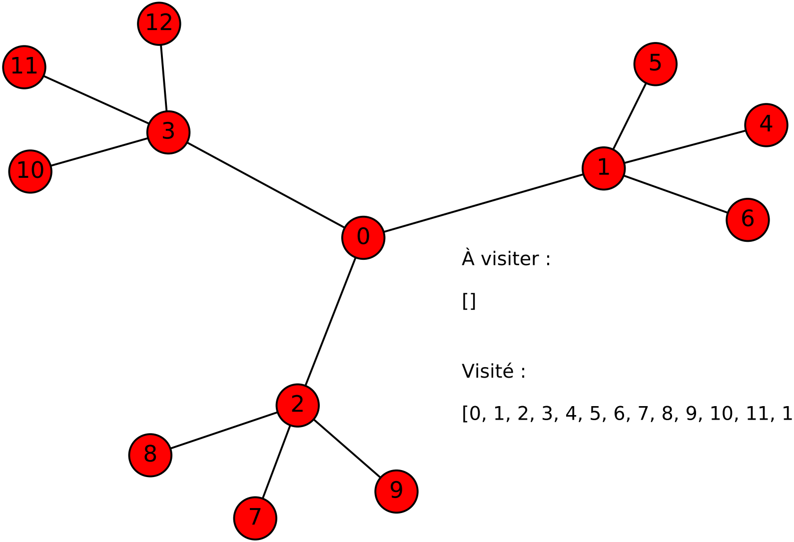


À visiter :

[12]

Visité :

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

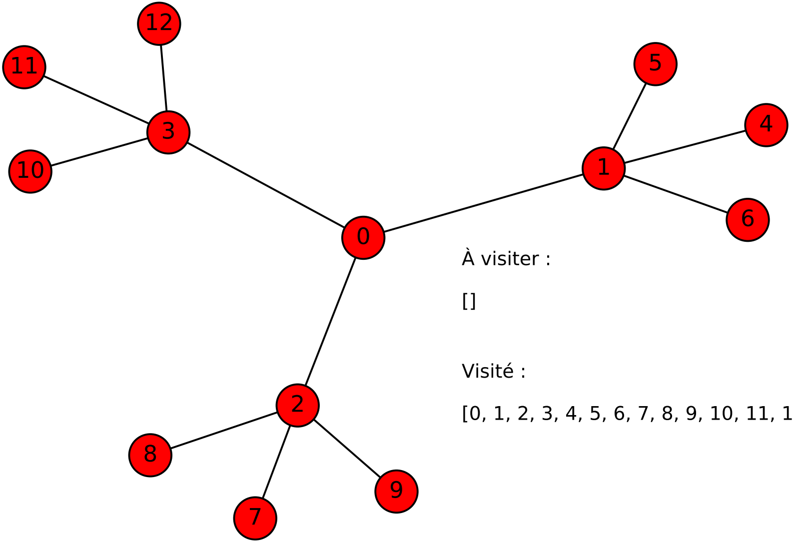


À visiter :

[]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12



À visiter :

[]

Visité :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

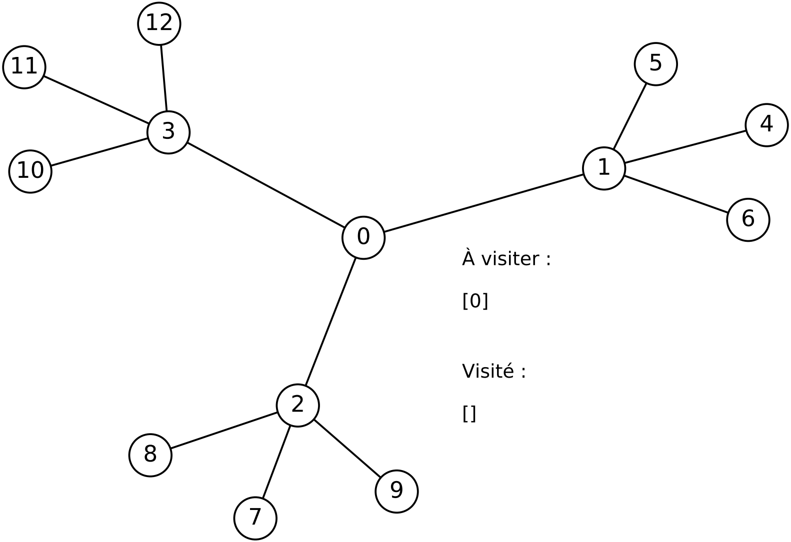
Dans un parcours en profondeur, on continue à avancer tant que c'est possible.

Tant que `a_visiter` n'est pas vide :

- $s =$ dernier élément de `a_visiter` ;
- retirer s de `a_visiter` ;
- rajouter s dans `visité` ;
- rajouter les voisins $adj(s)$ **à la fin de** `a_visiter`.

a_visiter fonctionne en "last in, first out" (LIFO) : les derniers éléments ajoutés sont les premiers à être explorés.

On continue dans la direction initiale tant que c'est possible.

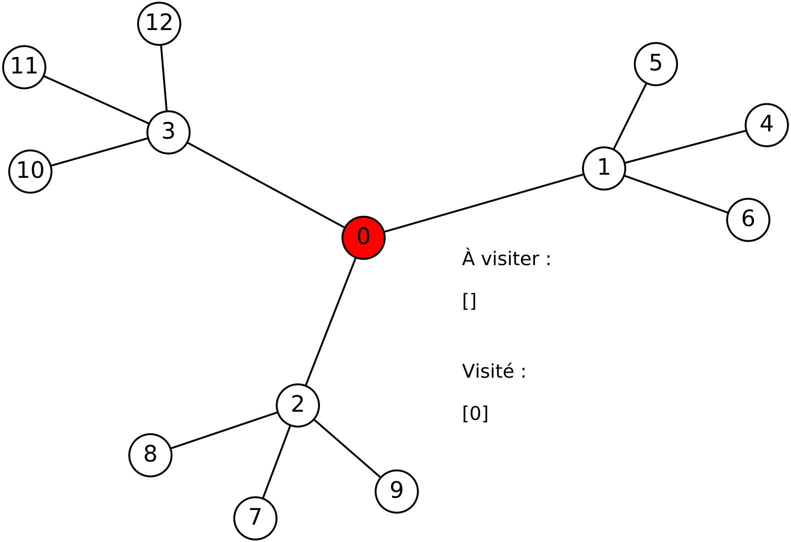


À visiter :

[0]

Visité :

[]

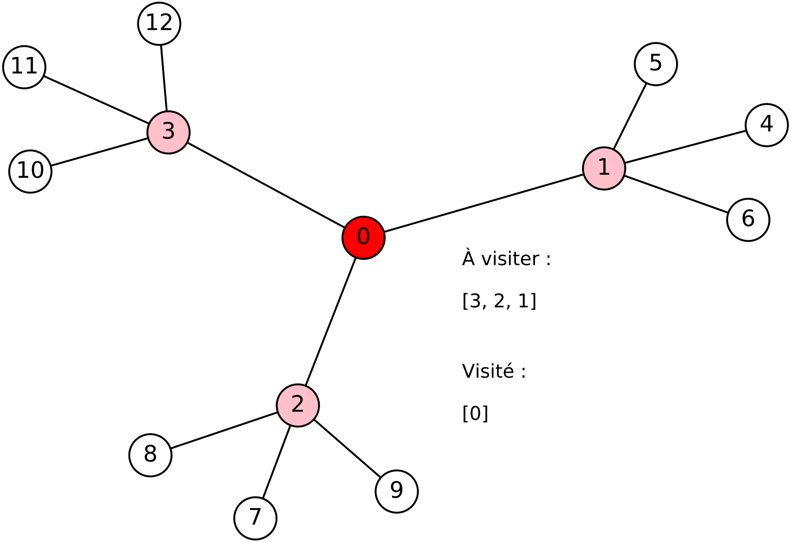


À visiter :

[]

Visité :

[0]

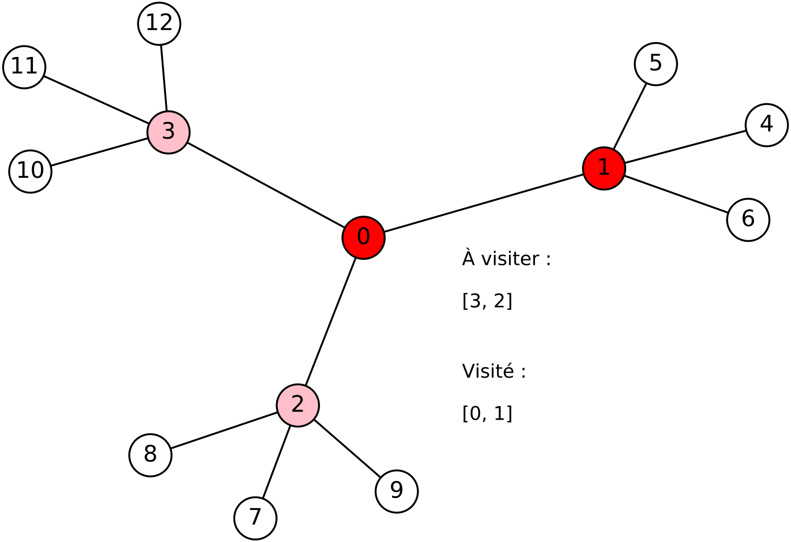


À visiter :

[3, 2, 1]

Visité :

[0]

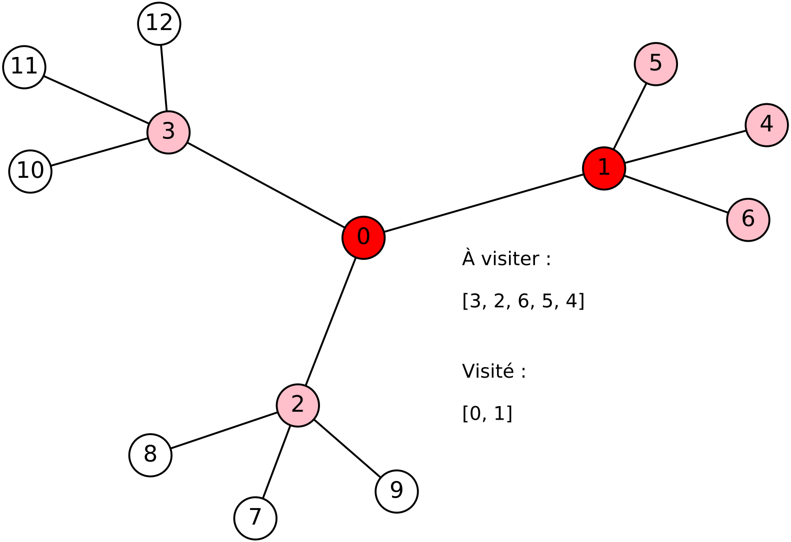


À visiter :

[3, 2]

Visité :

[0, 1]

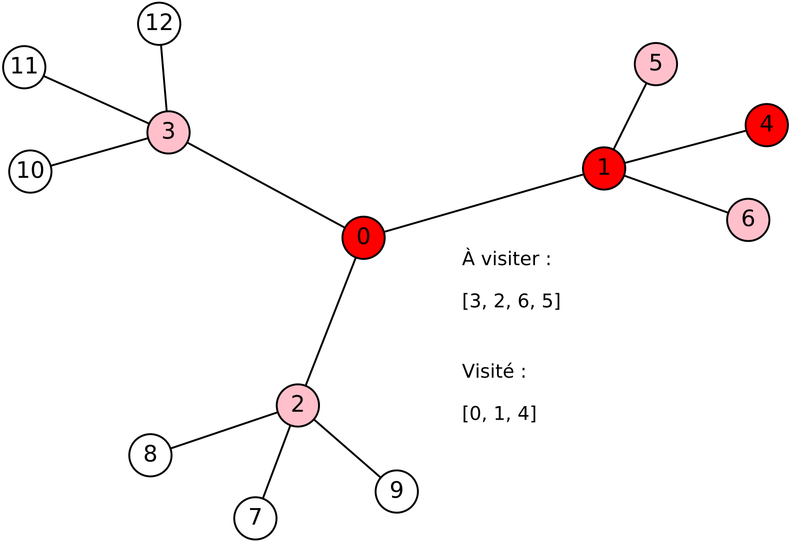


À visiter :

[3, 2, 6, 5, 4]

Visité :

[0, 1]

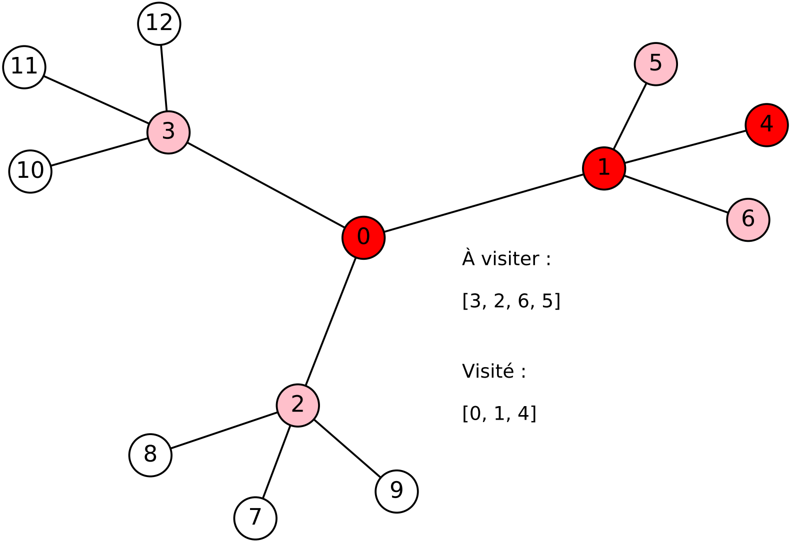


À visiter :

[3, 2, 6, 5]

Visité :

[0, 1, 4]

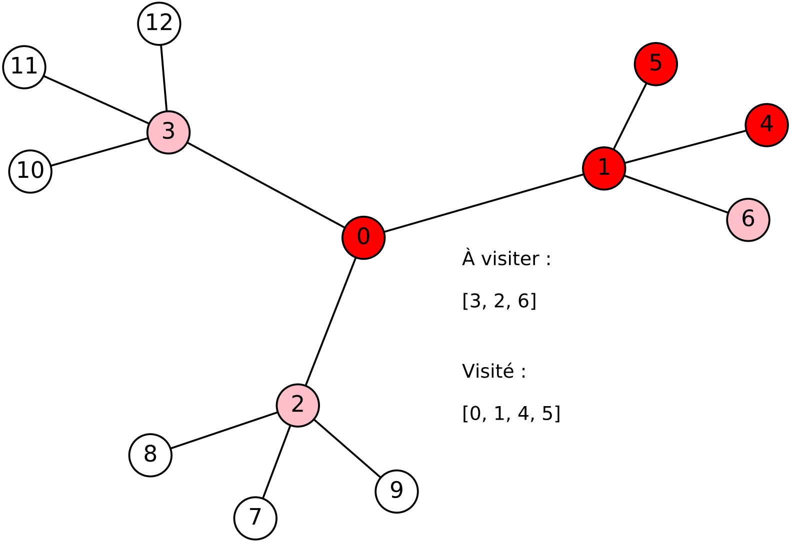


À visiter :

[3, 2, 6, 5]

Visité :

[0, 1, 4]

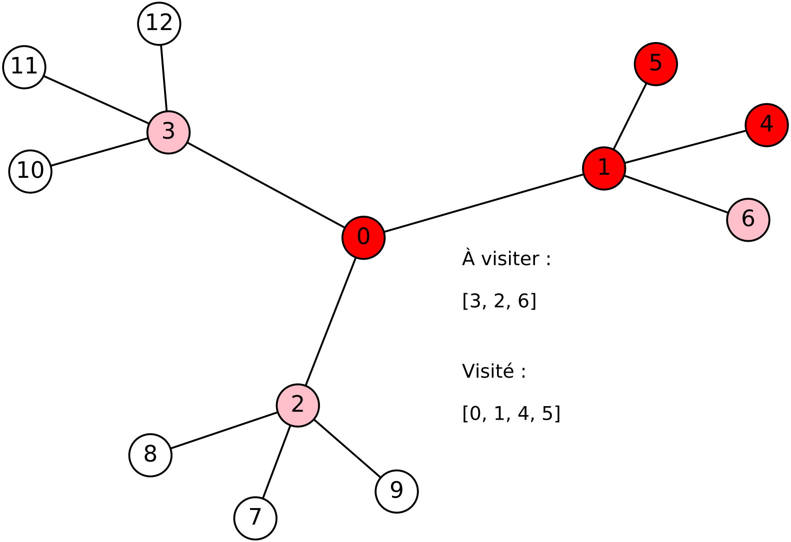


À visiter :

[3, 2, 6]

Visité :

[0, 1, 4, 5]

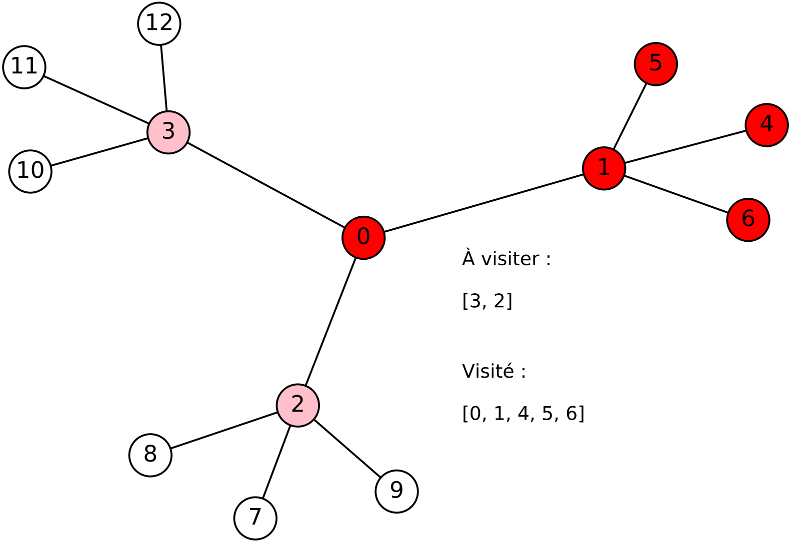


À visiter :

[3, 2, 6]

Visité :

[0, 1, 4, 5]

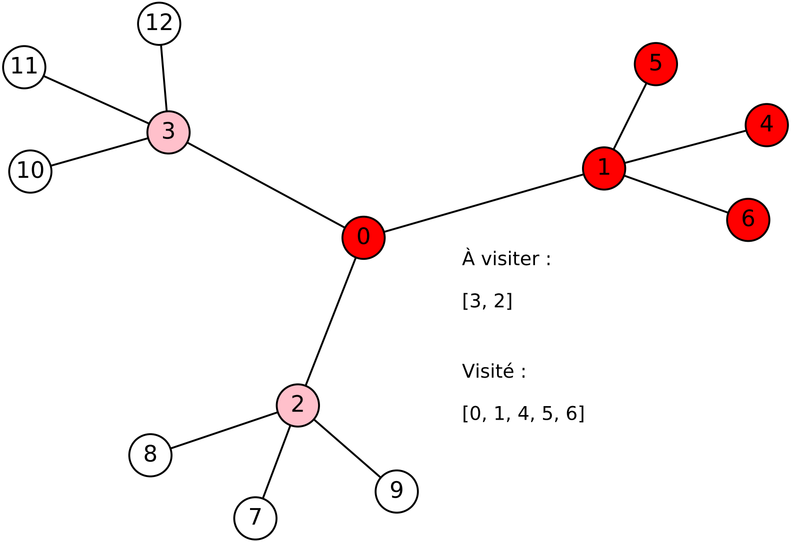


À visiter :

[3, 2]

Visité :

[0, 1, 4, 5, 6]

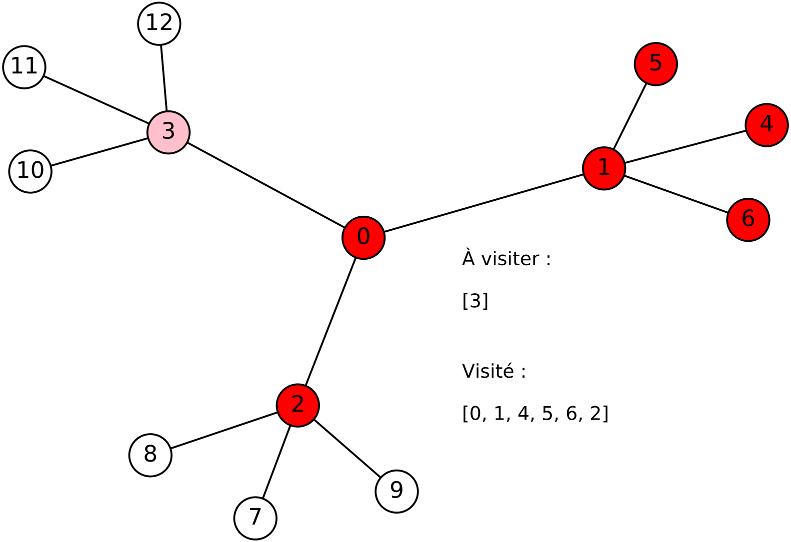


À visiter :

[3, 2]

Visité :

[0, 1, 4, 5, 6]

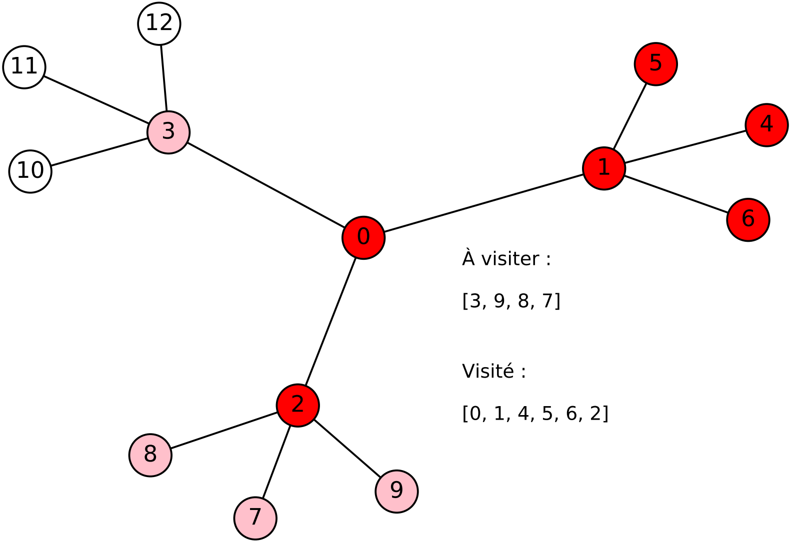


À visiter :

[3]

Visité :

[0, 1, 4, 5, 6, 2]

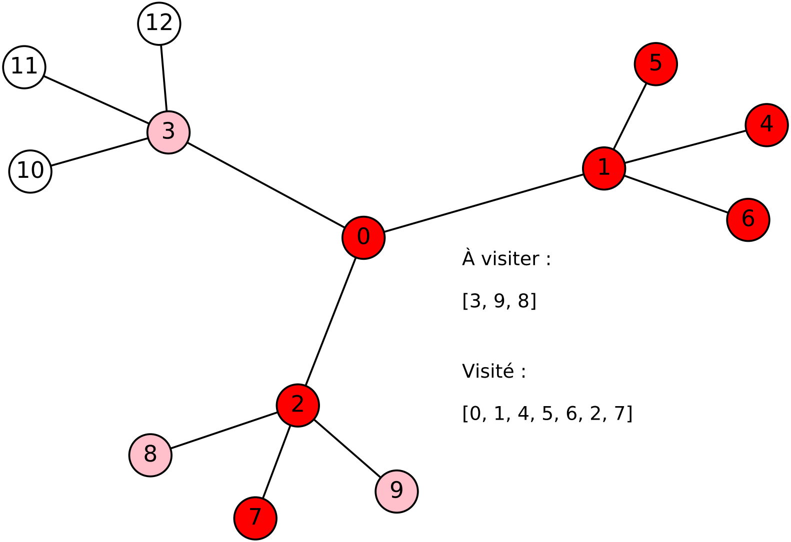


À visiter :

[3, 9, 8, 7]

Visité :

[0, 1, 4, 5, 6, 2]

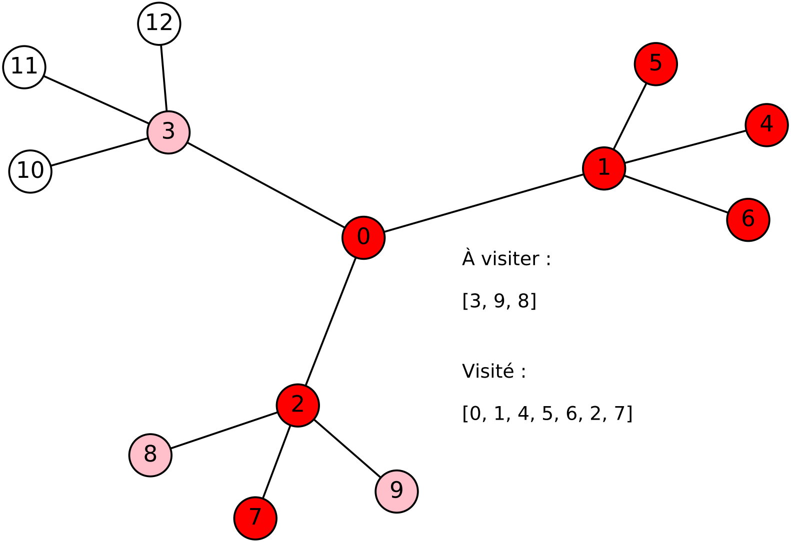


À visiter :

[3, 9, 8]

Visité :

[0, 1, 4, 5, 6, 2, 7]

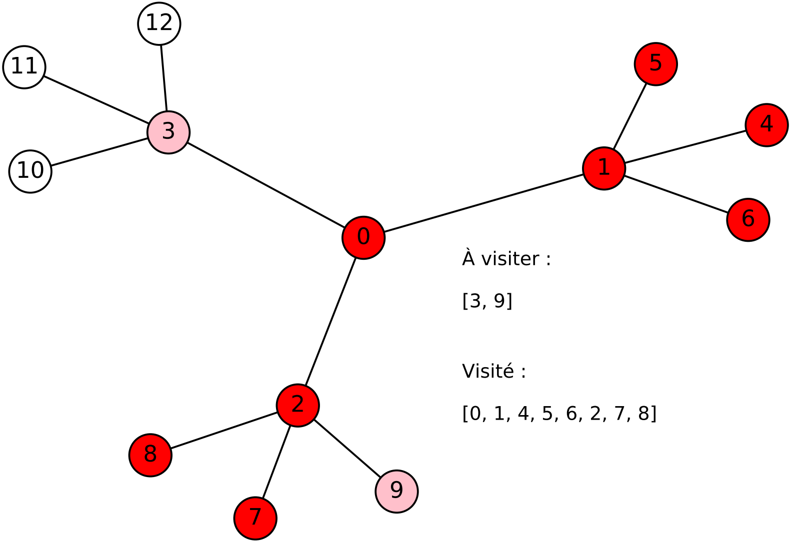


À visiter :

[3, 9, 8]

Visité :

[0, 1, 4, 5, 6, 2, 7]

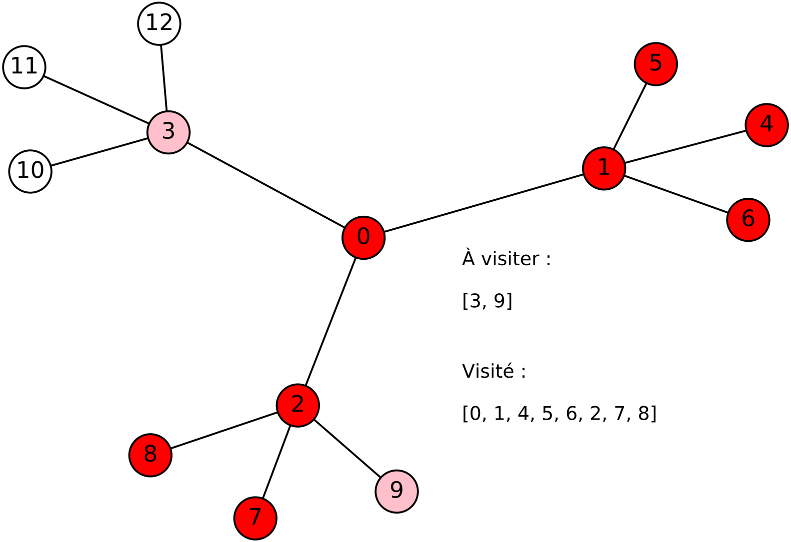


À visiter :

[3, 9]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8]

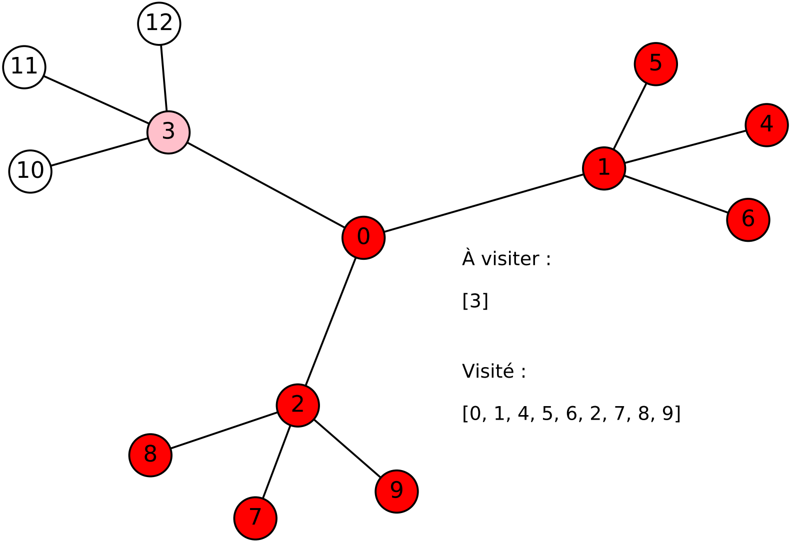


À visiter :

[3, 9]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8]

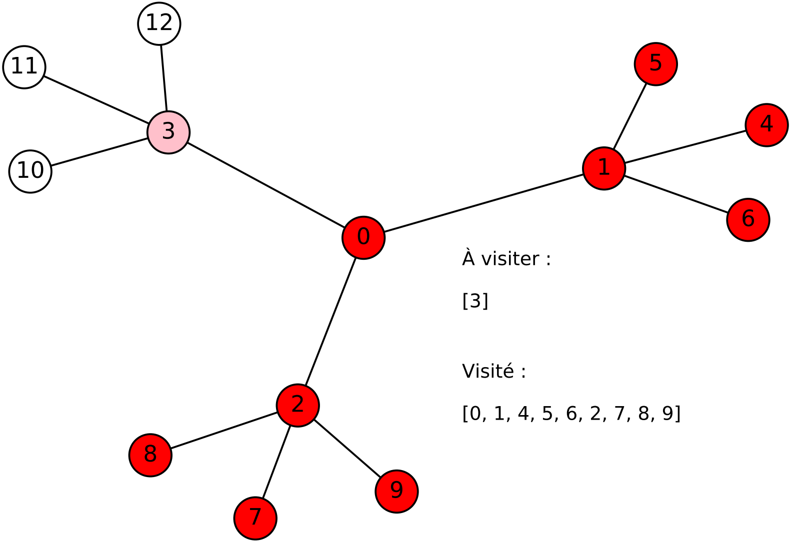


À visiter :

[3]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8, 9]

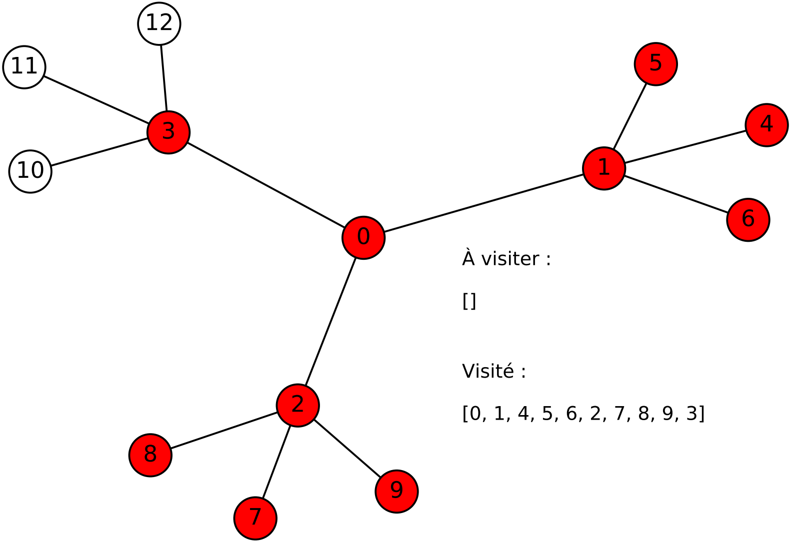


À visiter :

[3]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8, 9]

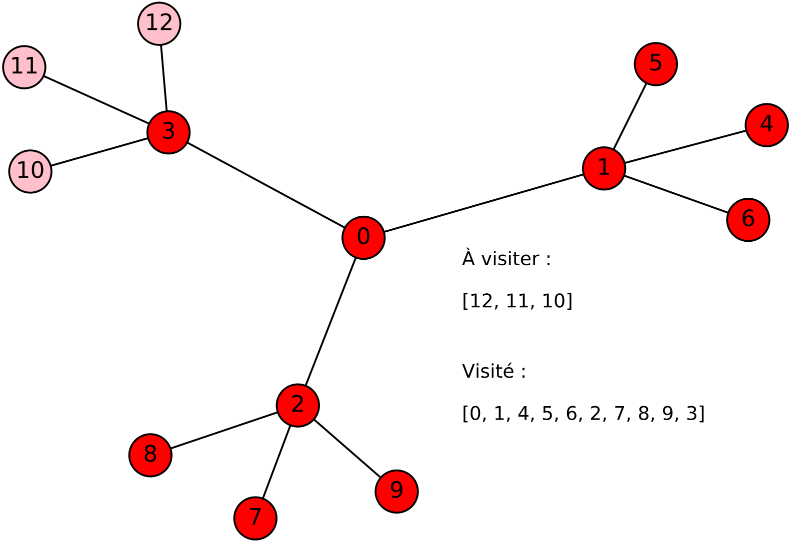


À visiter :

[]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8, 9, 3]

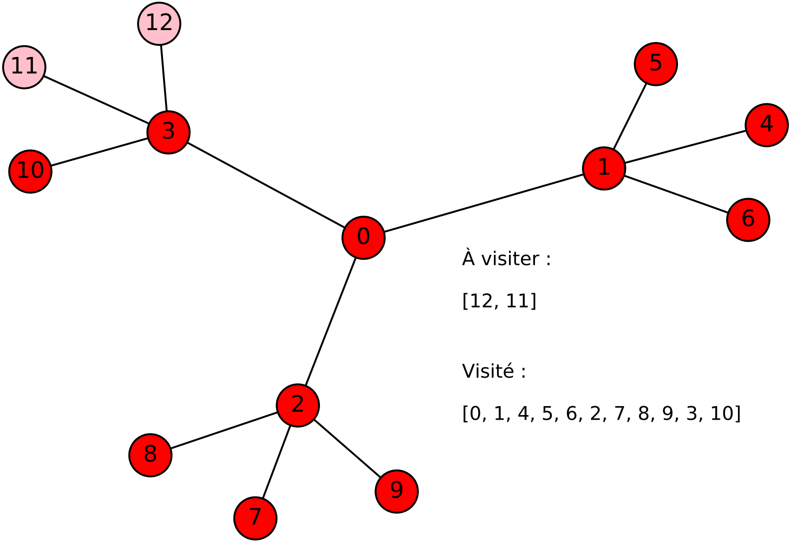


À visiter :

[12, 11, 10]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8, 9, 3]

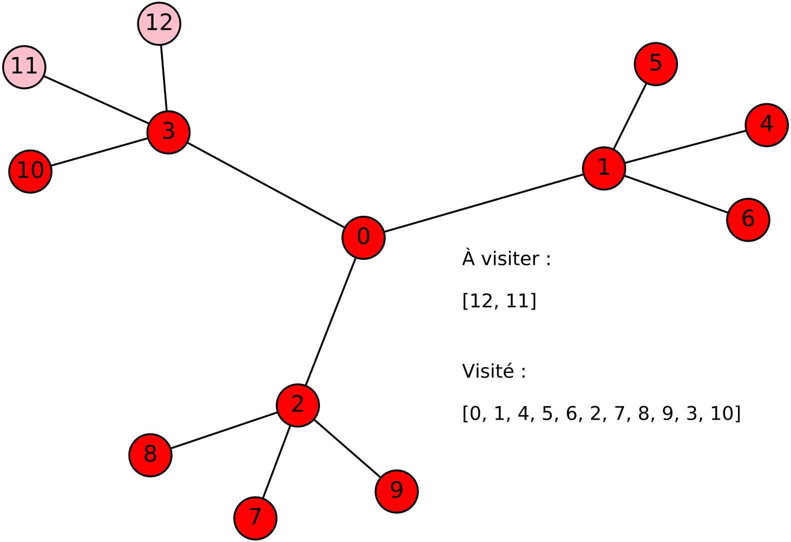


À visiter :

[12, 11]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8, 9, 3, 10]

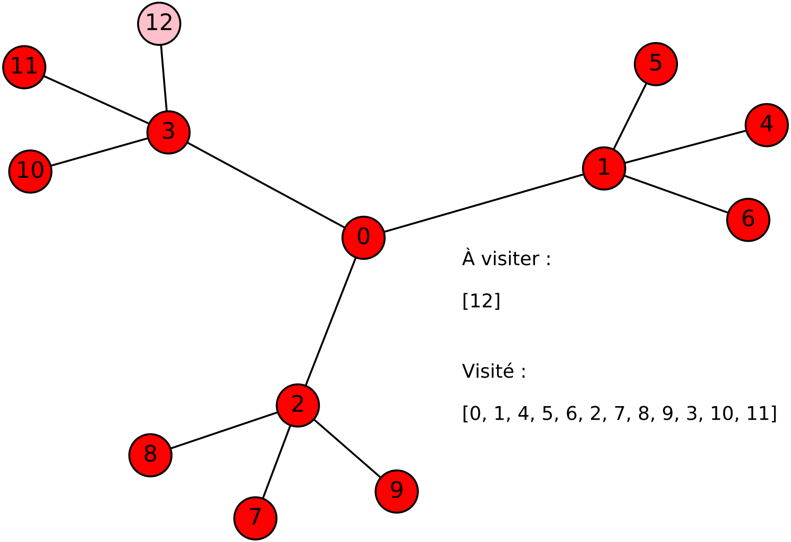


À visiter :

[12, 11]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8, 9, 3, 10]

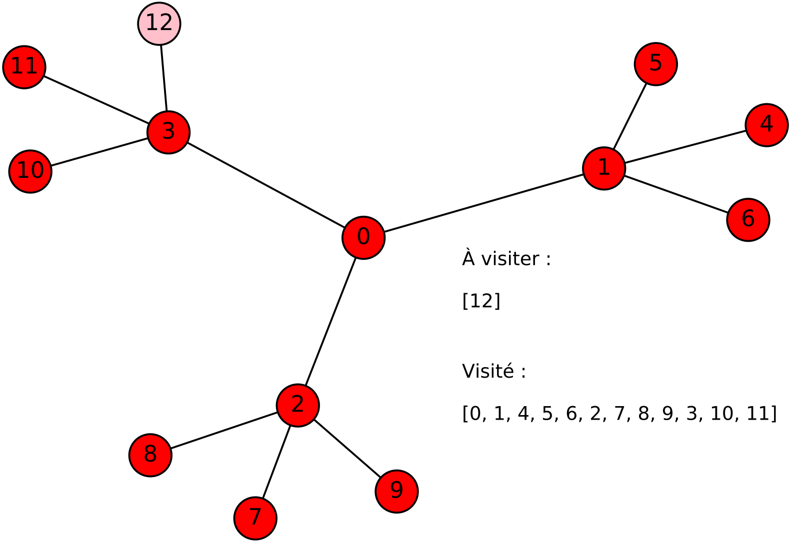


À visiter :

[12]

Visité :

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

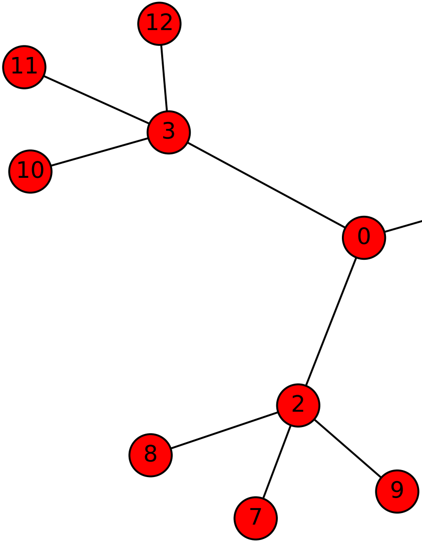


À visiter :

[12]

Visité :

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

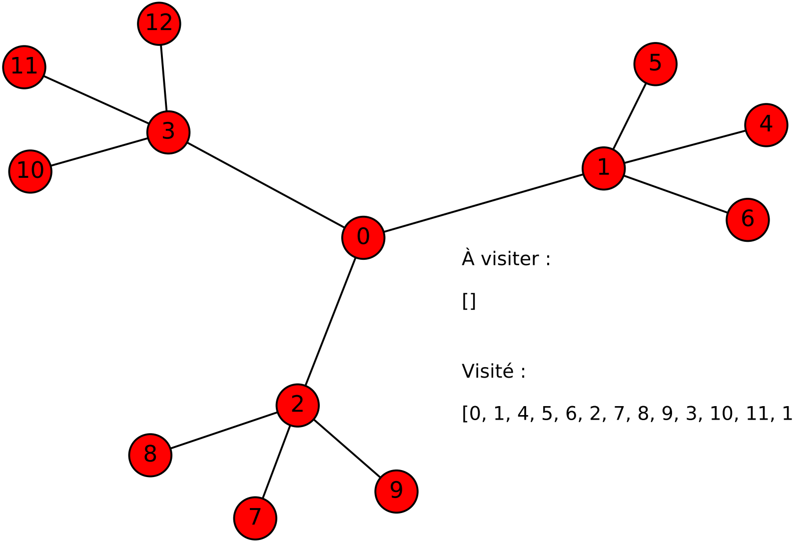


À visiter :

[]

Visité :

[0, 1, 4, 5, 6, 2, 7, 8, 9, 3, 10, 11, 12



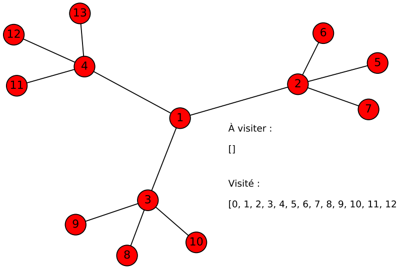
À visiter :

[]

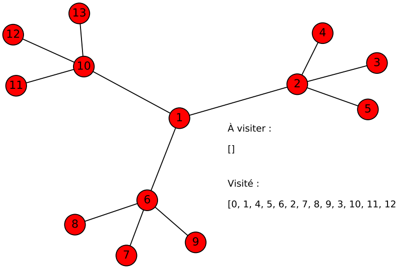
Visité :

[0, 1, 4, 5, 6, 2, 7, 8, 9, 3, 10, 11, 12

Comparons les graphes obtenus :



(a) Largeur



(b) Profondeur

Le parcours en **largeur** permet de trouver **les chemins les plus courts** entre un sommet et les autres.

Le parcours en **profondeur** permet de trouver **s'il existe un chemin** entre un sommet et les autres.

On va parler de ça en TP.

TP !

TP !

Dans le TP, on va :

- implémenter le parcours en largeur ;
- l'utiliser pour trouver des chemins ;
- application au labyrinthe ;
- un algorithme plus efficace ?

Ce TP est à rendre pour vendredi prochain avant le cours, pour que je puisse voir où vous en êtes.