S54MA2M7 : Informatique 2
Parcours de graphes

Luigi Santocanale
LIS, Aix-Marseille Université

15 janvier 2019

Plan

Plan

Definition

Un graphe (non-orienté) est un couple (V, E) où :

- V est un ensemble fini de sommets ;
- $E \subseteq P_2(V)$ est un ensemble d'arêtes.

Conventions :

- $n = \mathrm{card}(V)$, $m = \mathrm{card}(E)$,
- on écrit $xy \in E$ au lieu de $\{x, y\} \in E$.

Forets et arbres

- (V, E) est *acyclique* si ...
- (V, E) est *connexe* si ...
- (V, E) est un *foret* si est acyclique ;
- (V, E) est un *arbre* si acyclique et connexe.

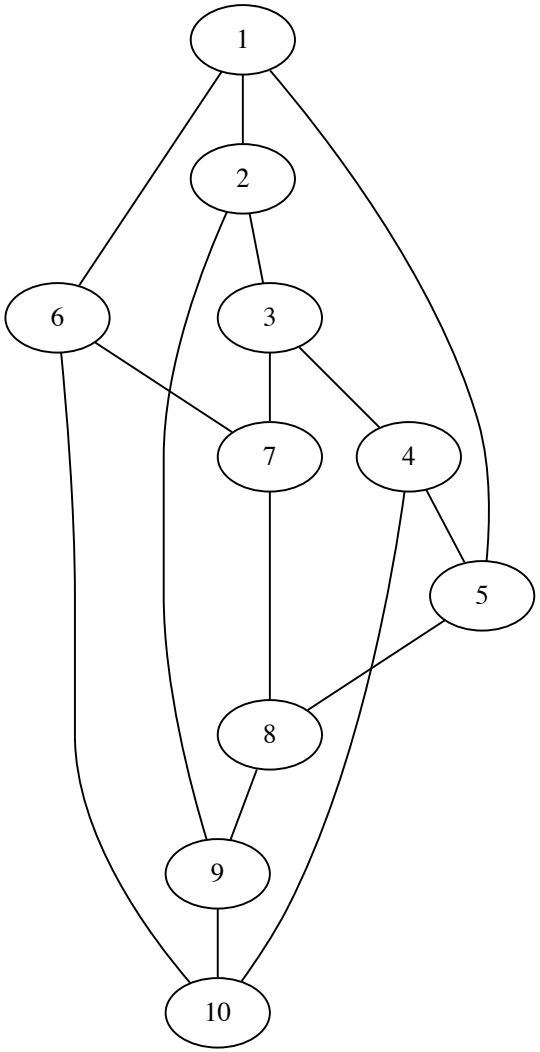
Lemma

Si (V, E) *est un arbre, alors* $m = n - 1$.

Foret/arbre recouvrant

- (V', E') est un *graphe partiel* de (V, E) si
  1. $V' = V$,
  2. $E' \subseteq E$,
- si (V, T) est un graphe partiel de (V, E) tel que (V, T) est un arbre (foret), on appelle (V, T) un arbre (foret) recouvrant de (V, E).

Représentation d'un graphe sur ordinateur par listes d'adjacence ou matrice d'adjacence



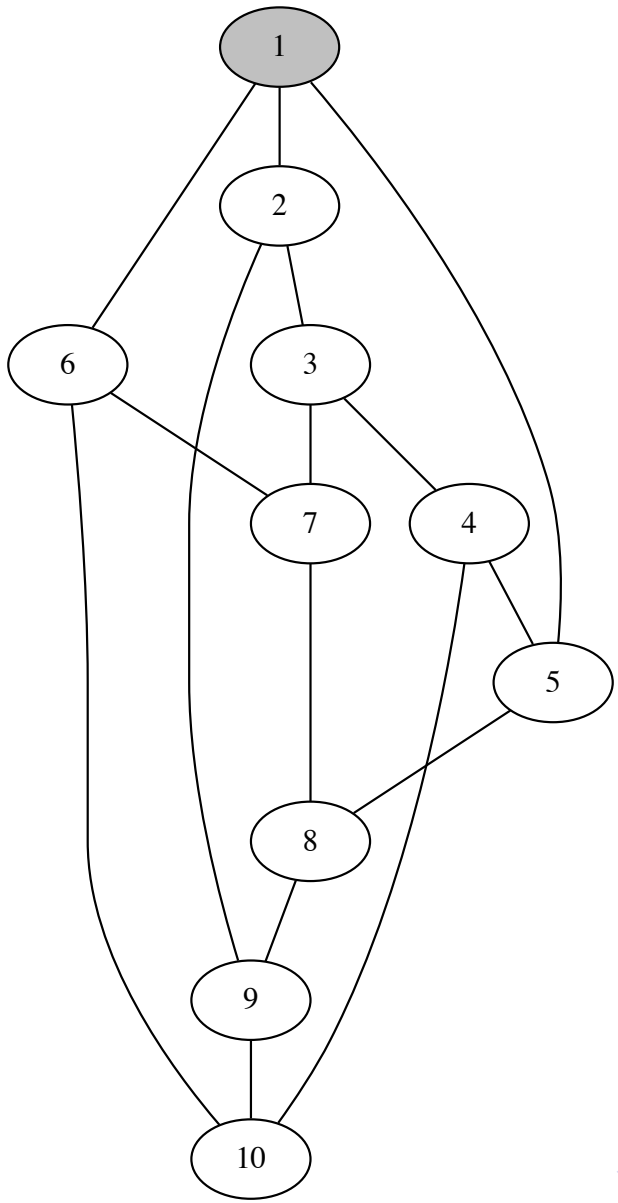

```
adjLists = {
    1: [2, 6, 5],
    2: [3, 9, 1],
    3: [4, 7, 2],
    4: [5, 10, 3],
    5: [1, 8, 4],
    6: [7, 10, 1],
    7: [8, 6, 3],
    8: [9, 7, 5],
    9: [10, 8, 2],
    10: [6, 9, 4]}

adjMatrix = array([
    [0, 1, 0, 0, 1, 1, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 1, 0],
    [0, 1, 0, 1, 0, 0, 1, 0, 0, 0],
    [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
    [1, 0, 0, 1, 0, 0, 0, 1, 0, 0],
    [1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
    [0, 0, 1, 0, 0, 1, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0, 1, 0, 1, 0],
    [0, 1, 0, 0, 0, 0, 0, 1, 0, 1],
    [0, 0, 0, 1, 0, 1, 0, 0, 1, 0]
  ])
```

Plan

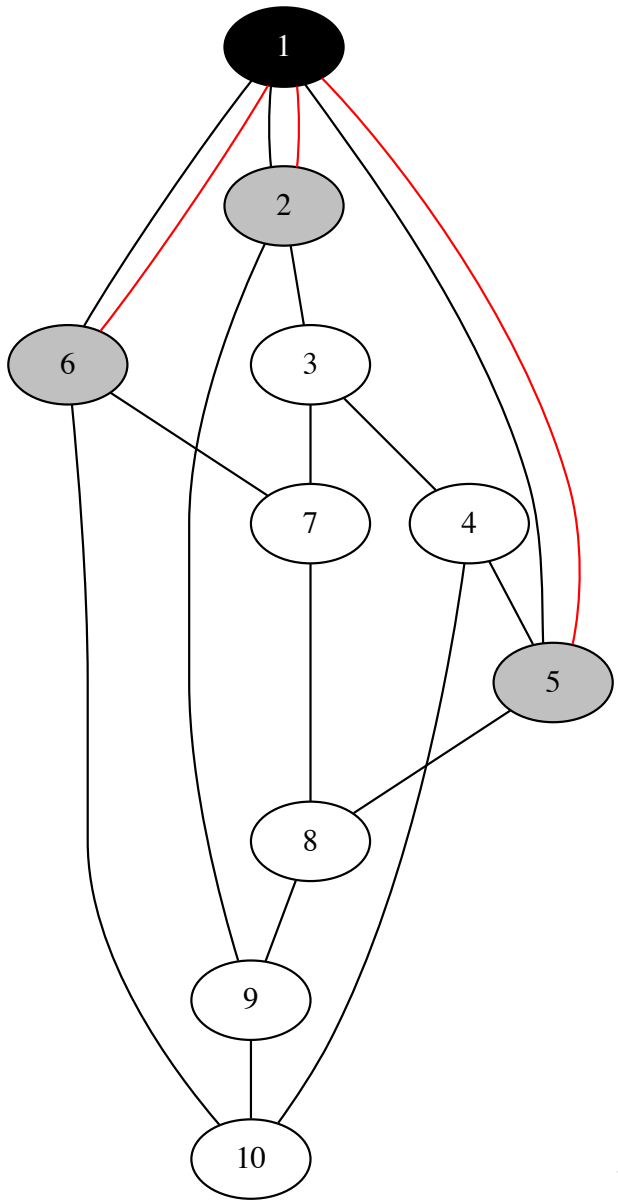
Parcours de graphe

```
def parcours(adjLists):
    vertices = list(adjLists.keys())
    color = {}
    parent = {}
    toBeTreated = set()  # x in toBeTreated iff color[x] == 'Gray'

    for x in vertices:
        color[x] = 'White'

    for x in vertices:
        if color[x] == 'White':
            color[x] = 'Gray'
            toBeTreated.add(x)
            parent[x] = None
            while toBeTreated != set():
                y = toBeTreated.pop()
                for z in adjLists[y]:
                    if color[z] == 'White':
                        color[z] = 'Gray'
                        toBeTreated.add(z)
                        parent[z] = y
                color[y] = 'Black'

    return parent
```

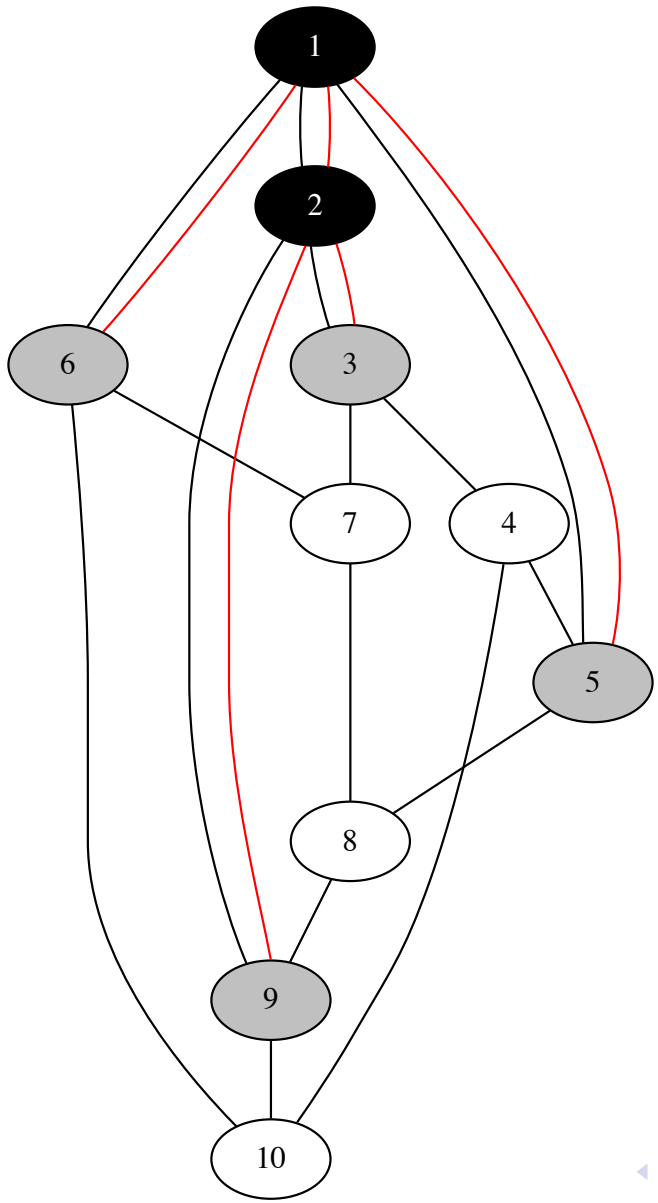
Exemple



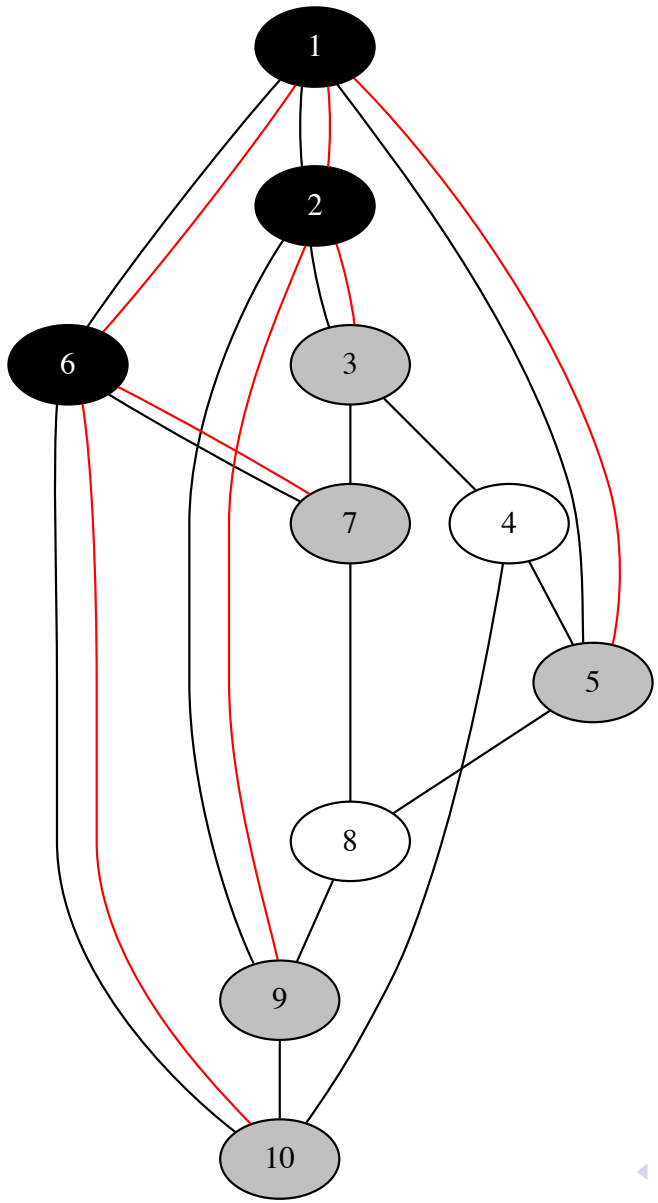
Exemple



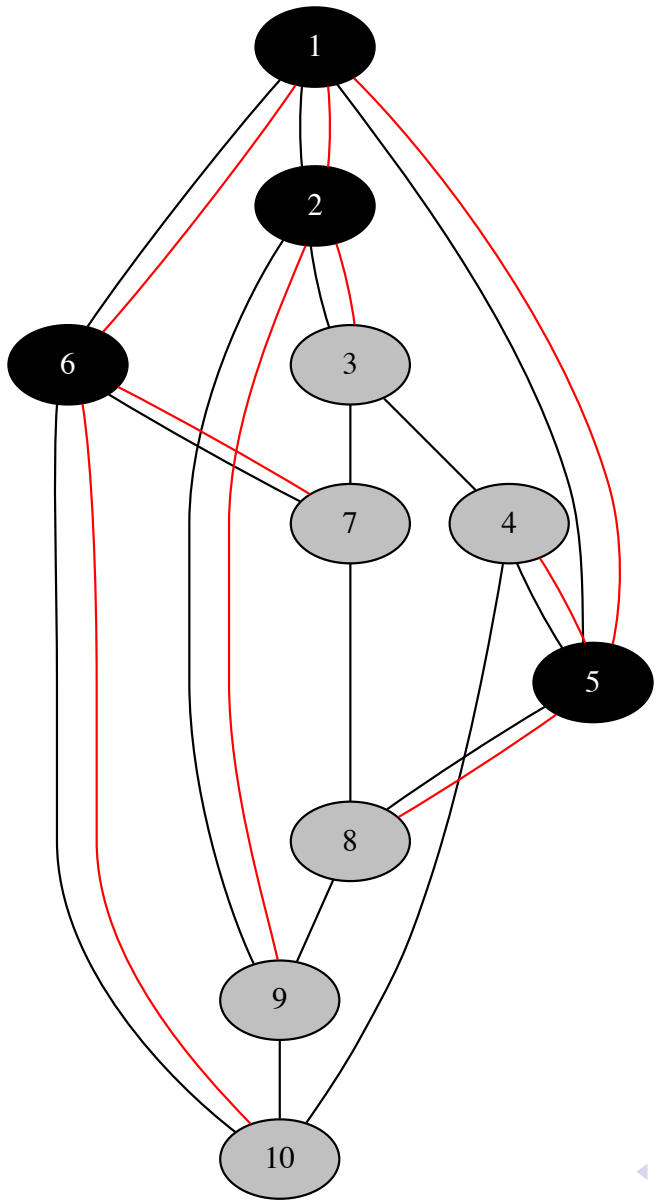
Exemple



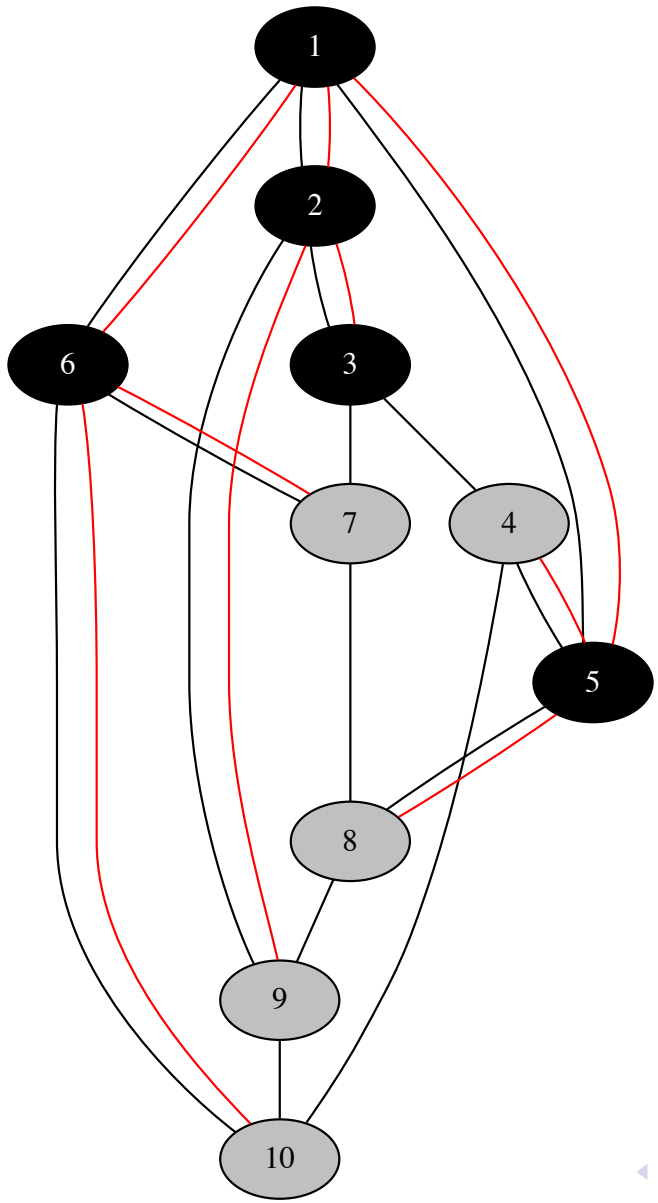
Exemple



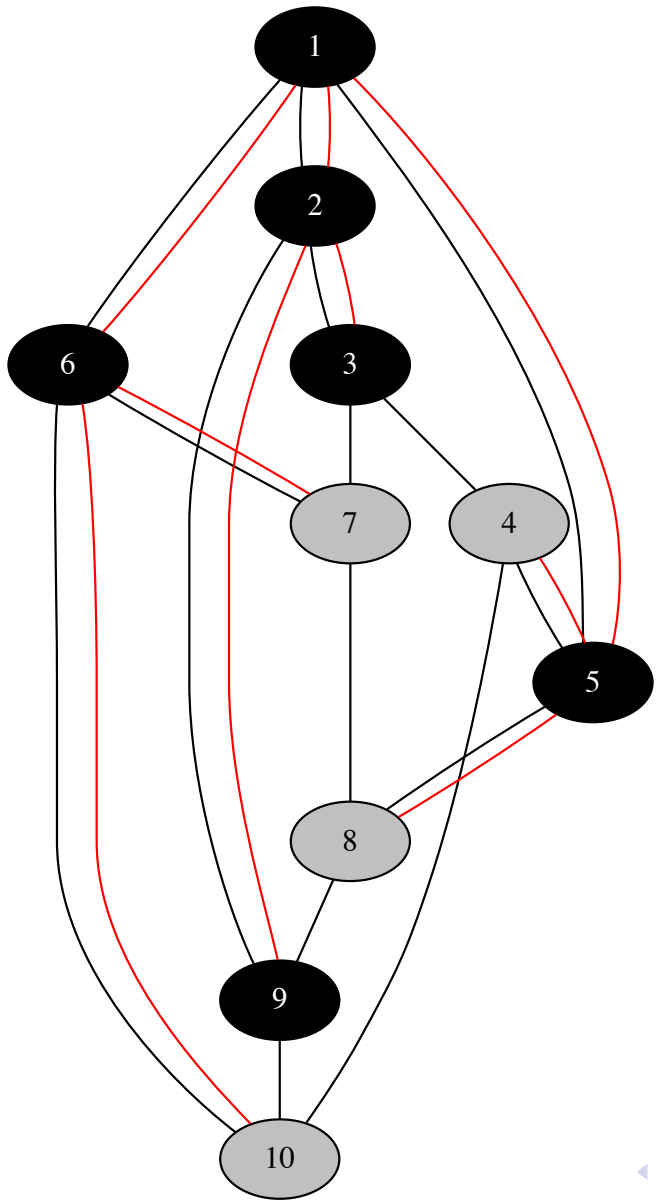
Exemple



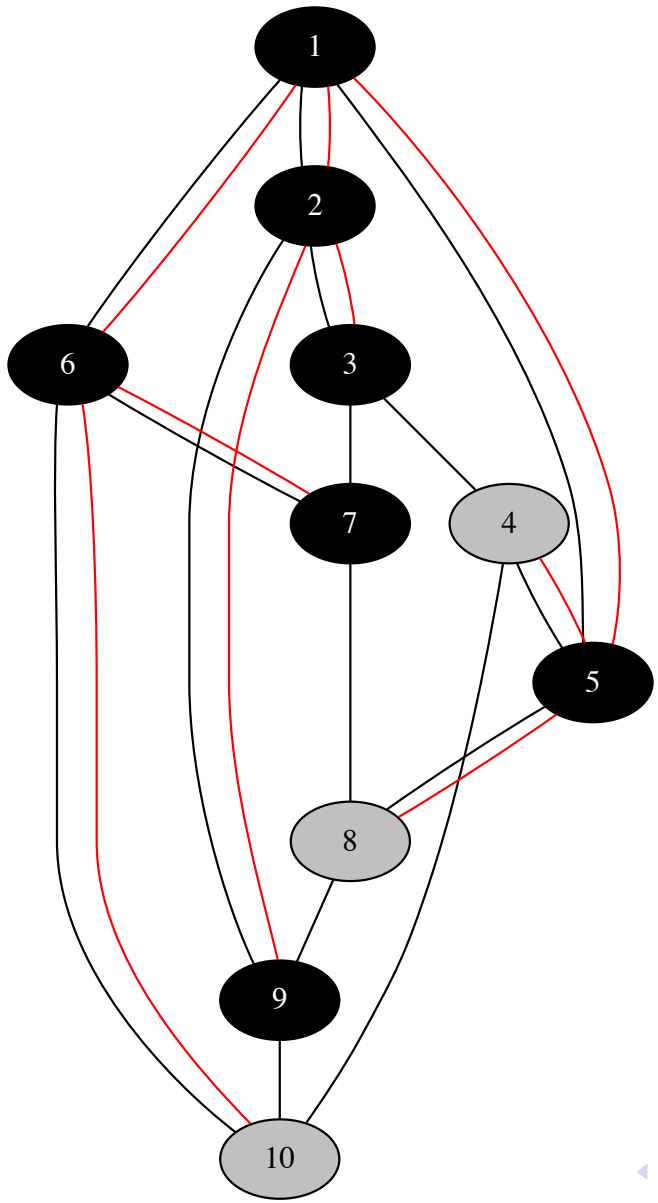
Exemple



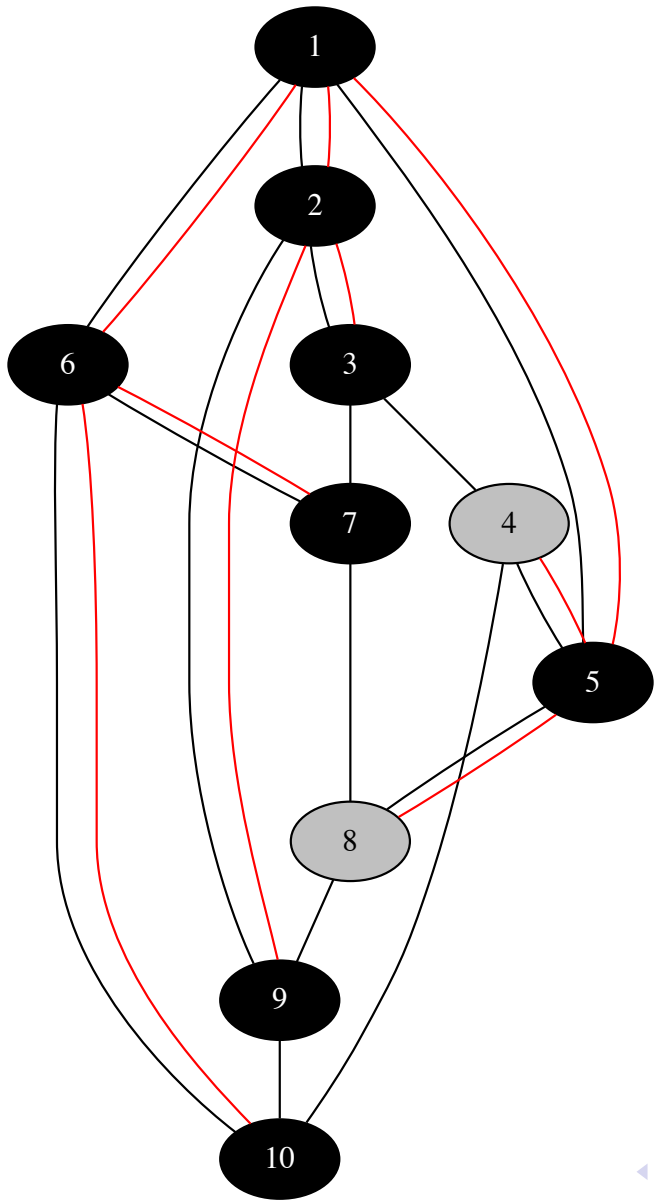
Exemple



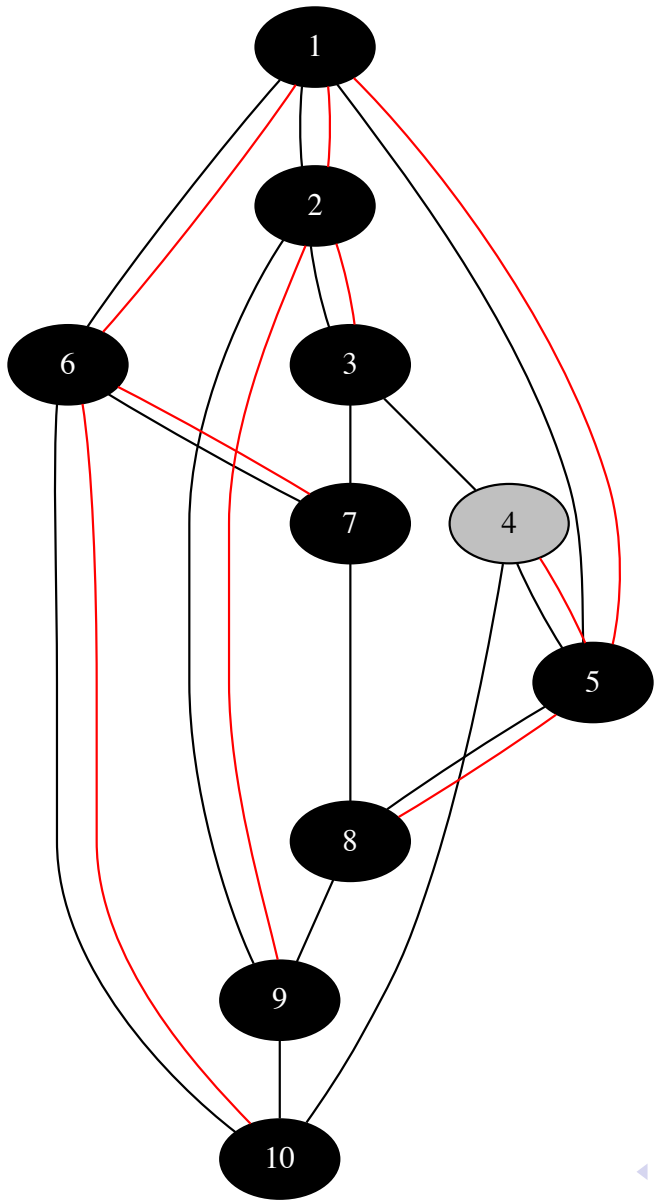
Exemple



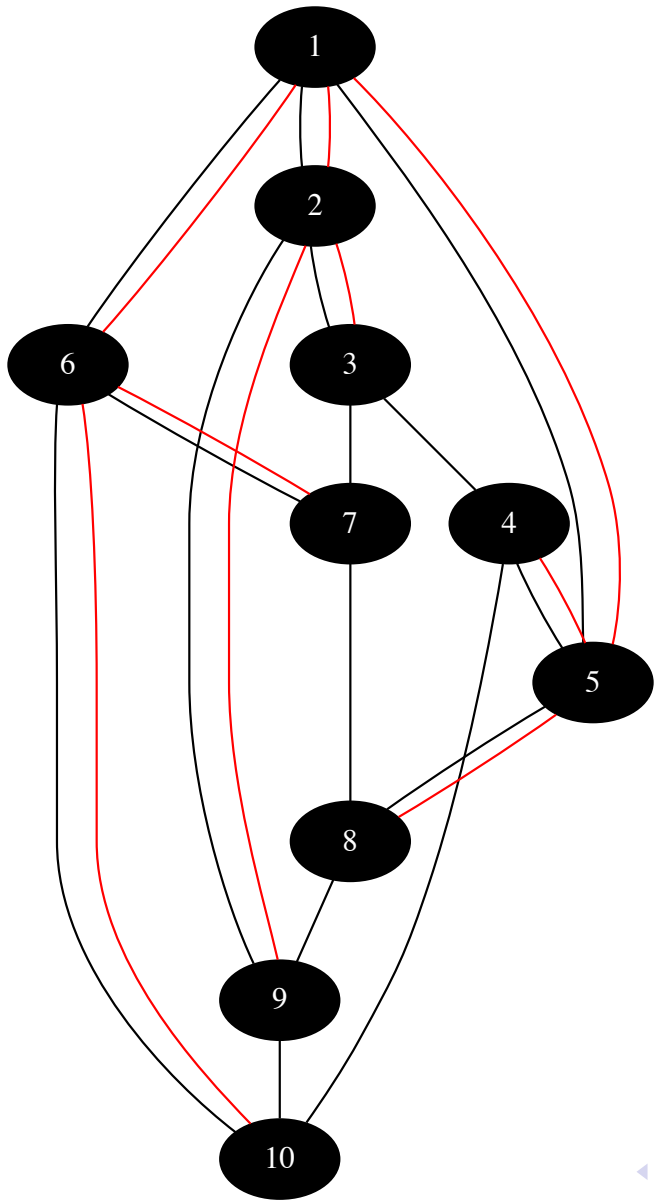
Exemple



Exemple



Exemple



Parcours en largeur (BFS)

`Set` est implementé comme un queue (`FIFO`) :

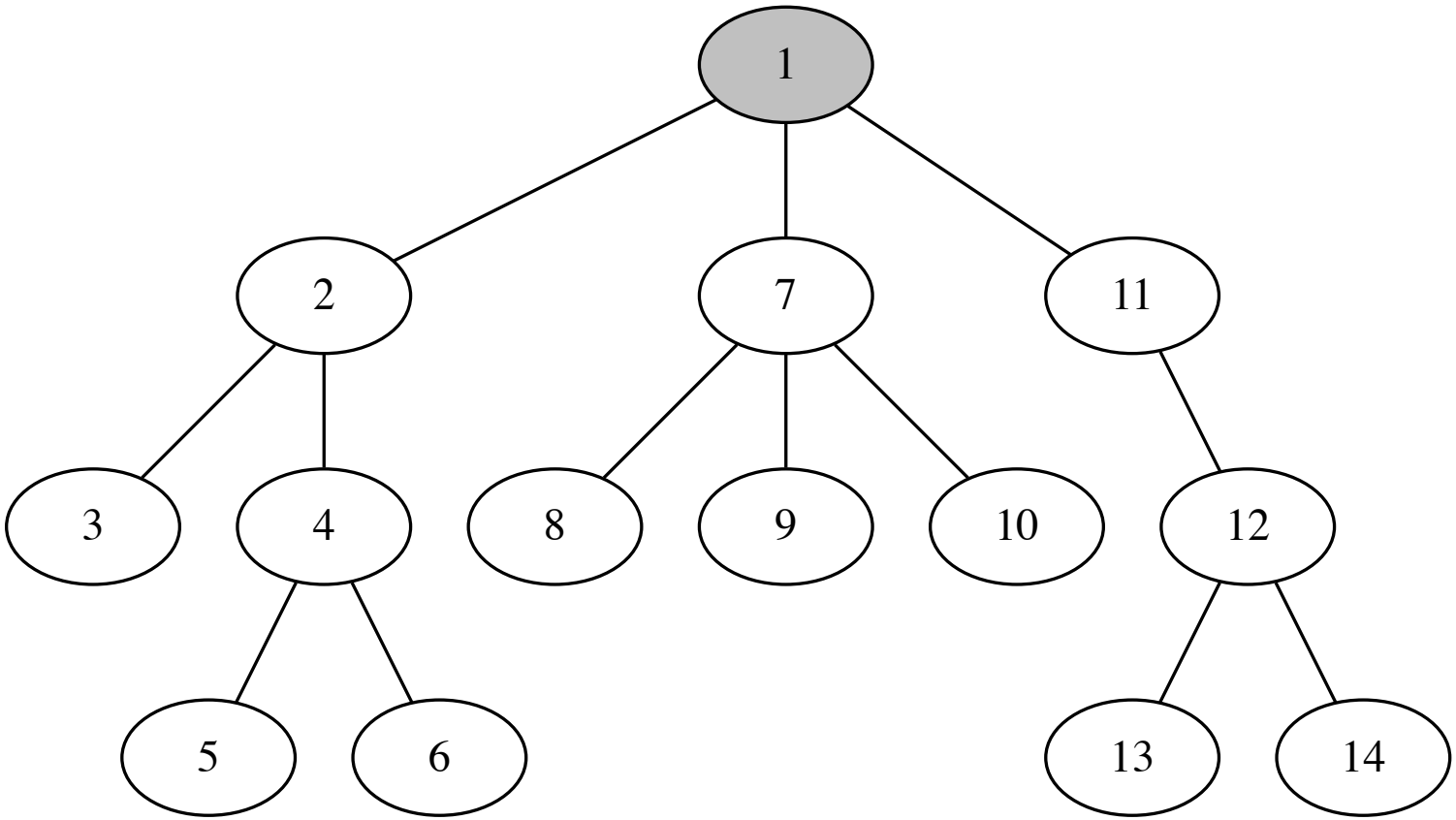
```
class Fifo(object):
    def __init__(self):
        self.state = []

    def isEmpty(self):
        return self.state == []

    def pop(self):
        if self.isEmpty():
            raise Exception('Empty Fifo')
        else:
            y = self.state.pop(0)
            return y

    def add(self, y):
        self.state.append(y)
```

Parcours en profondeur (DFS)

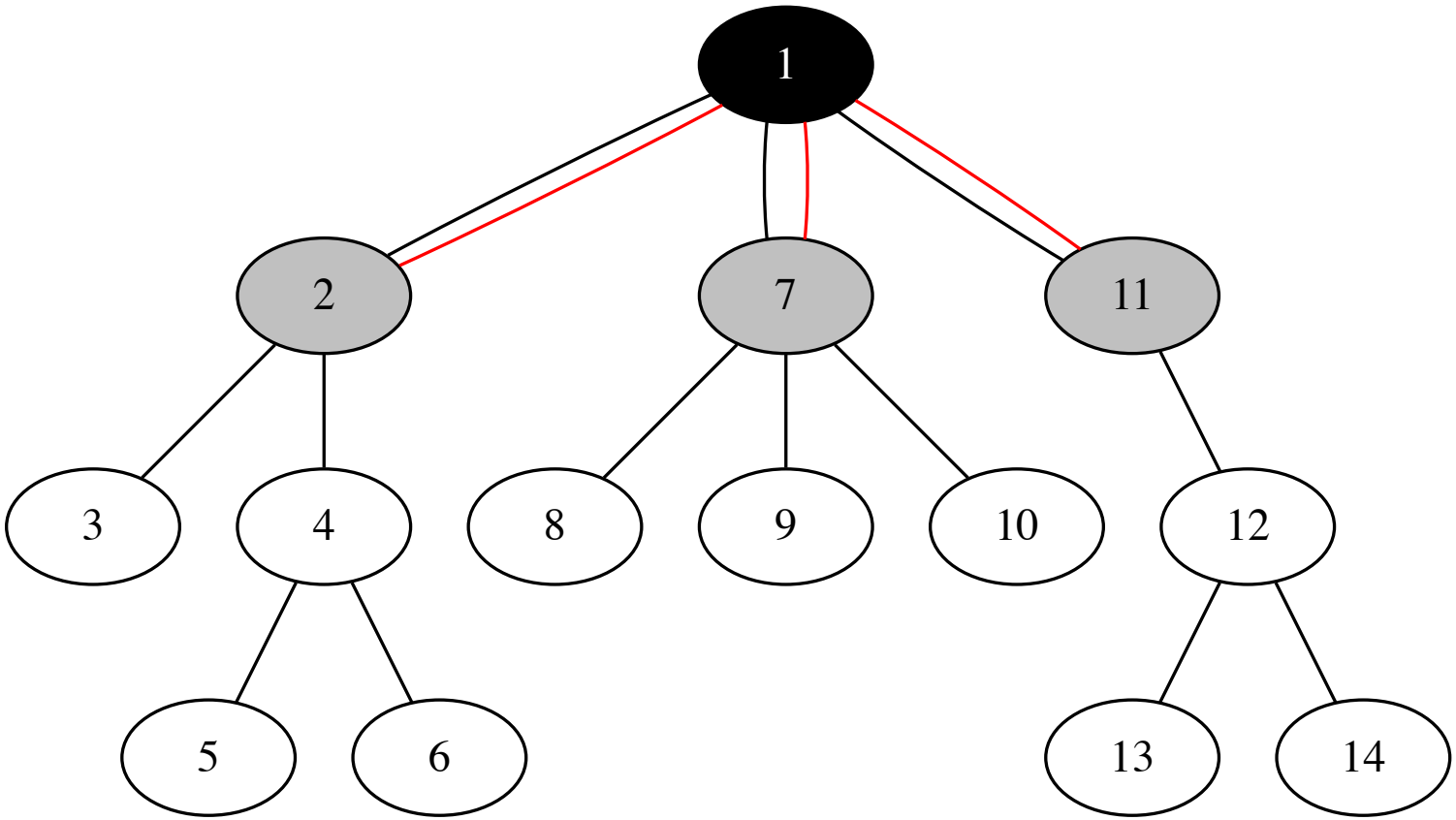
L'ensemble `toBeTreated` est implementé comme un `pile` (`LIFO`) :

```
class Stack(object):
    def __init__(self):
        self.state = []

    def isEmpty(self):
        return self.state == []

    def pop(self):
        if self.isEmpty():
            raise Exception('Empty Stack')
        else:
            y = self.state.pop()
            return y

    def add(self, y):
        self.state.append(y)
```

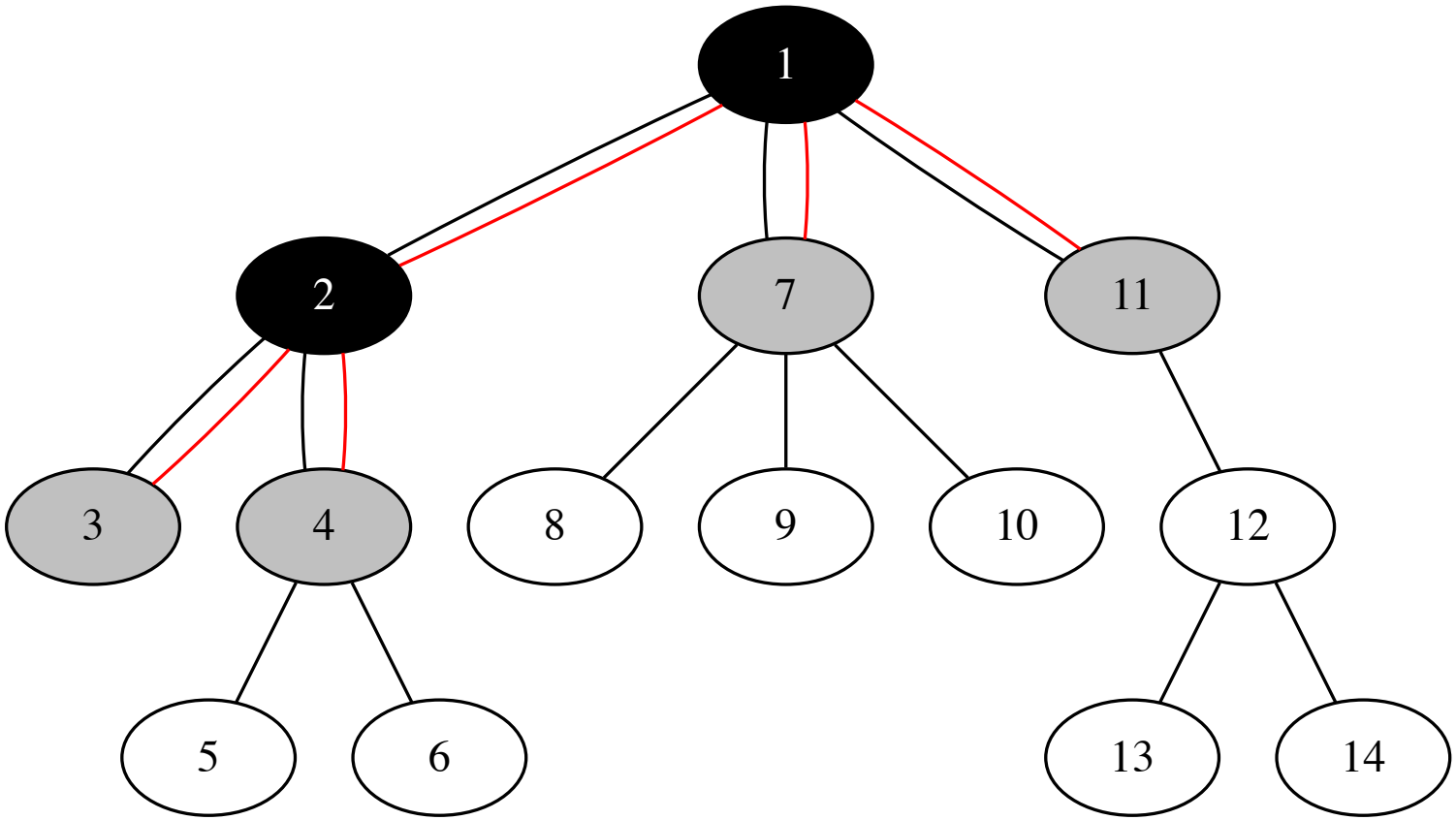
Parcours en largeur sur un arbre



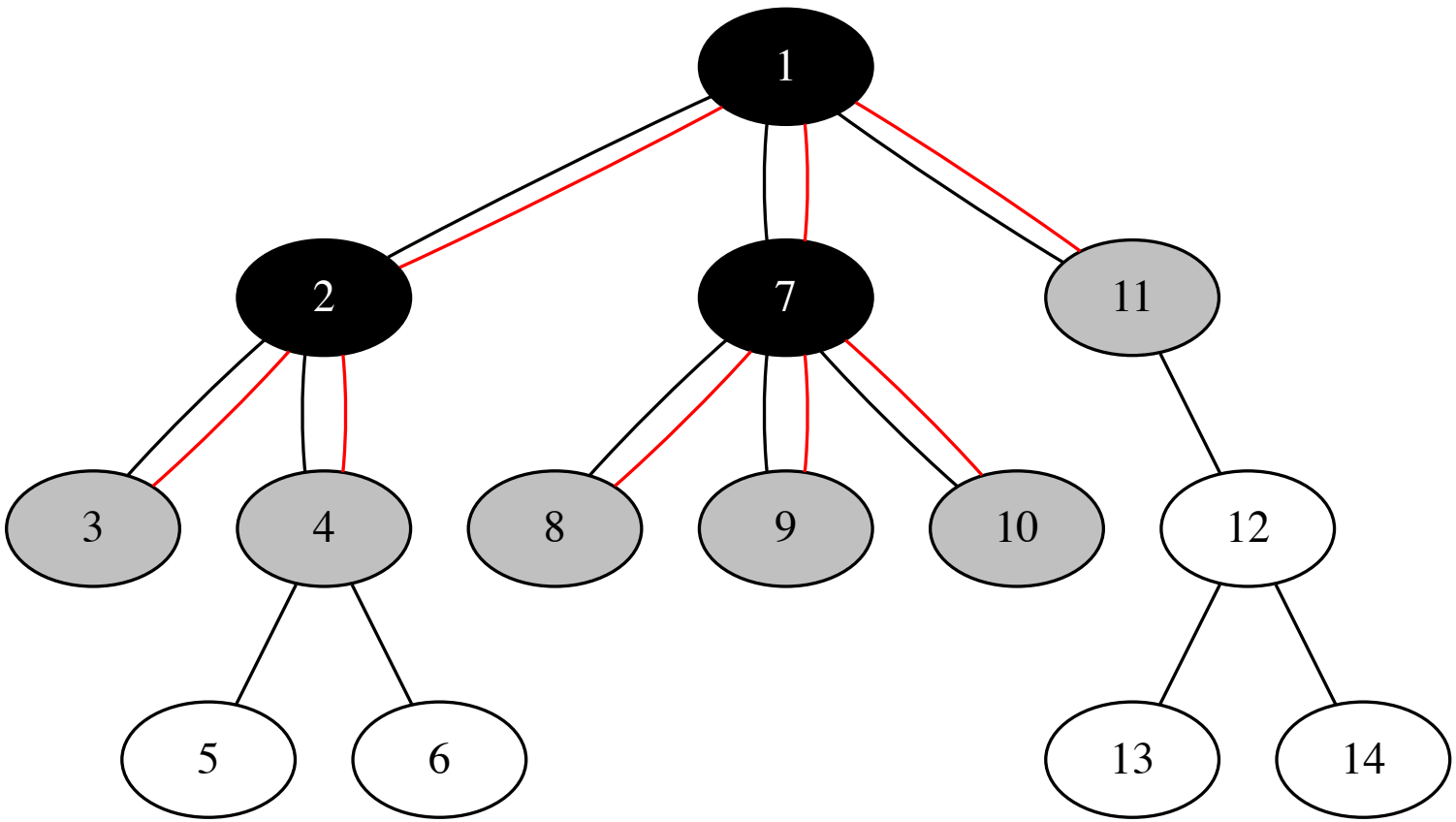
Parcours en largeur sur un arbre



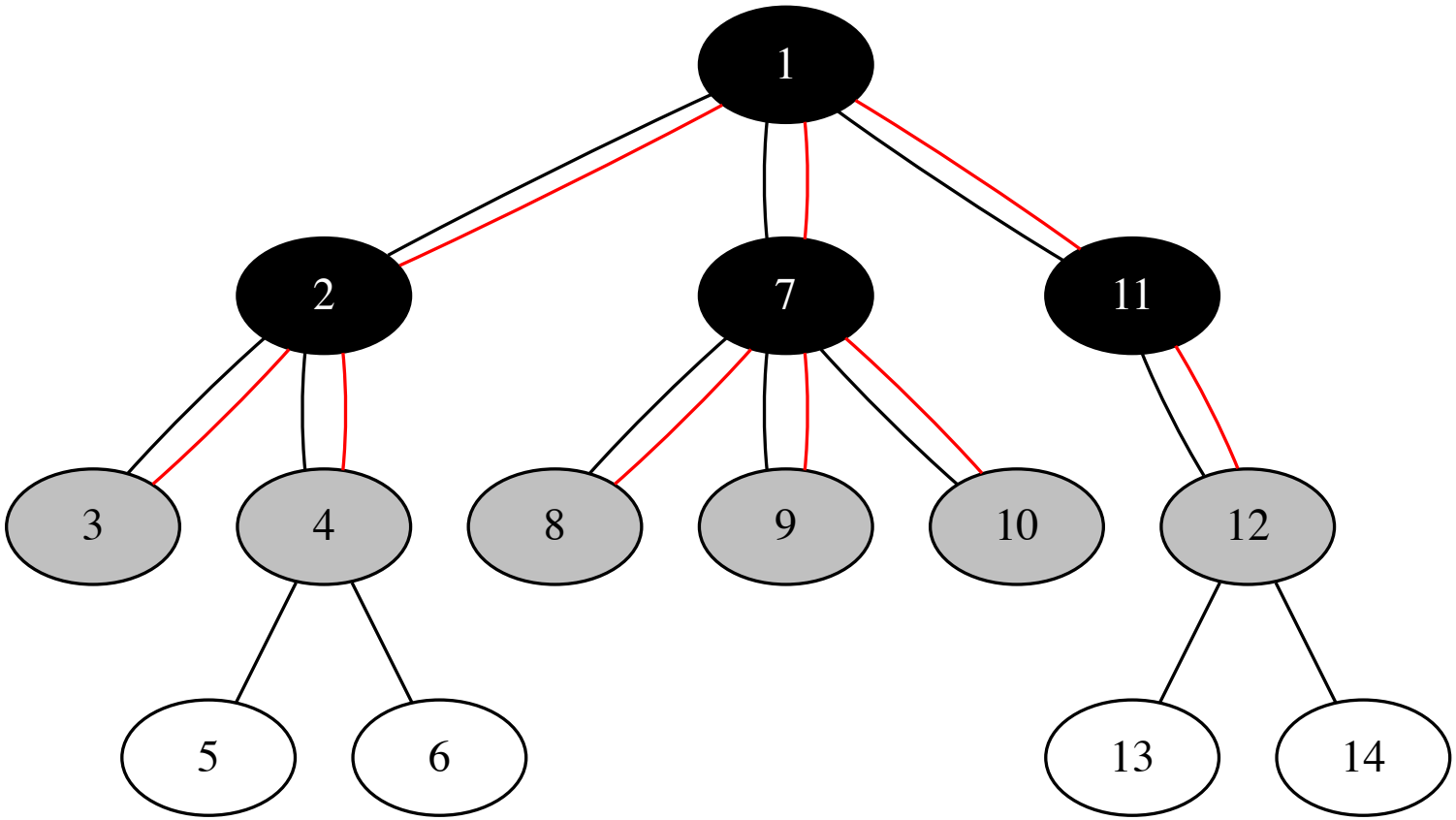
Parcours en largeur sur un arbre



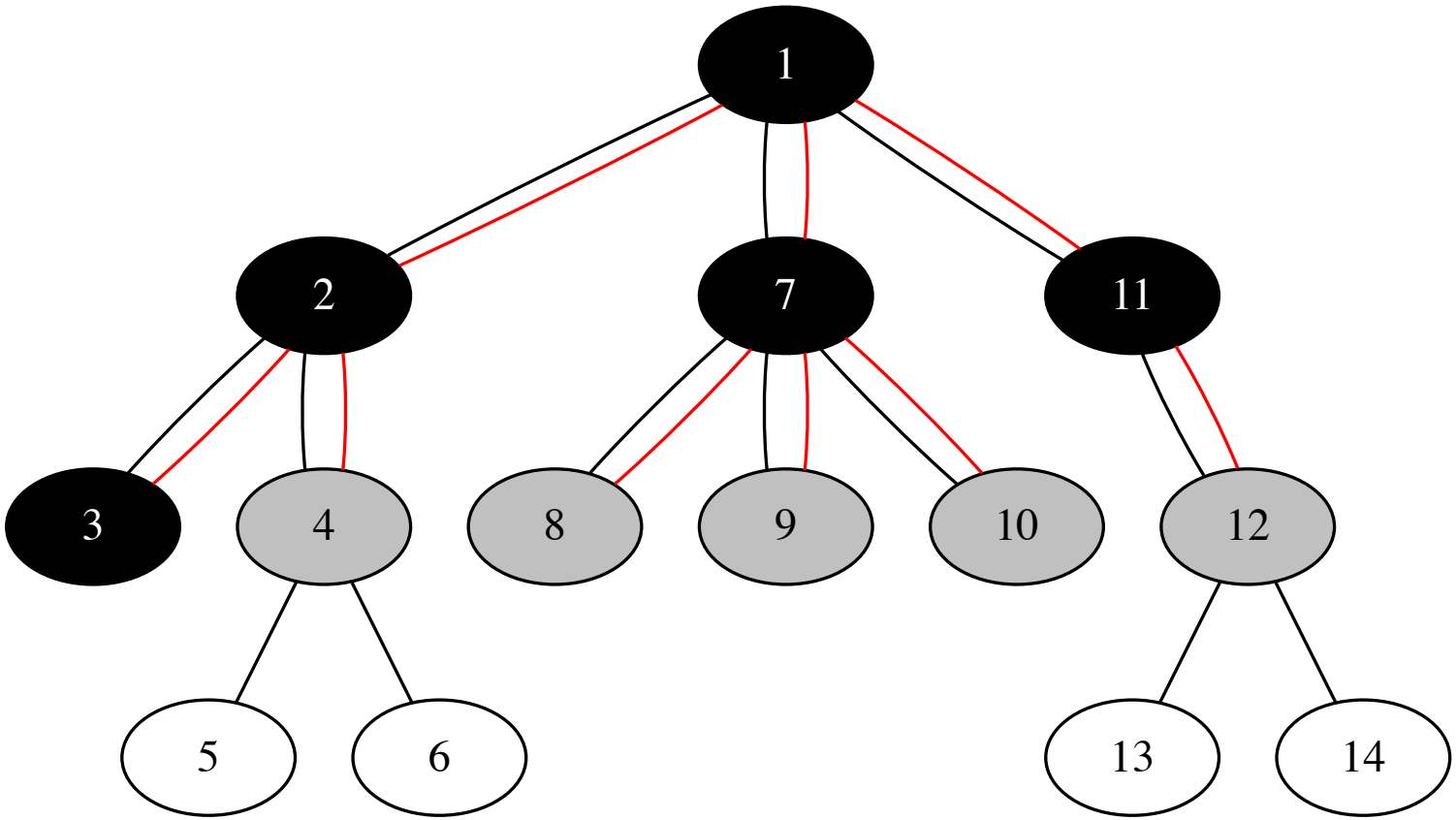
Parcours en largeur sur un arbre



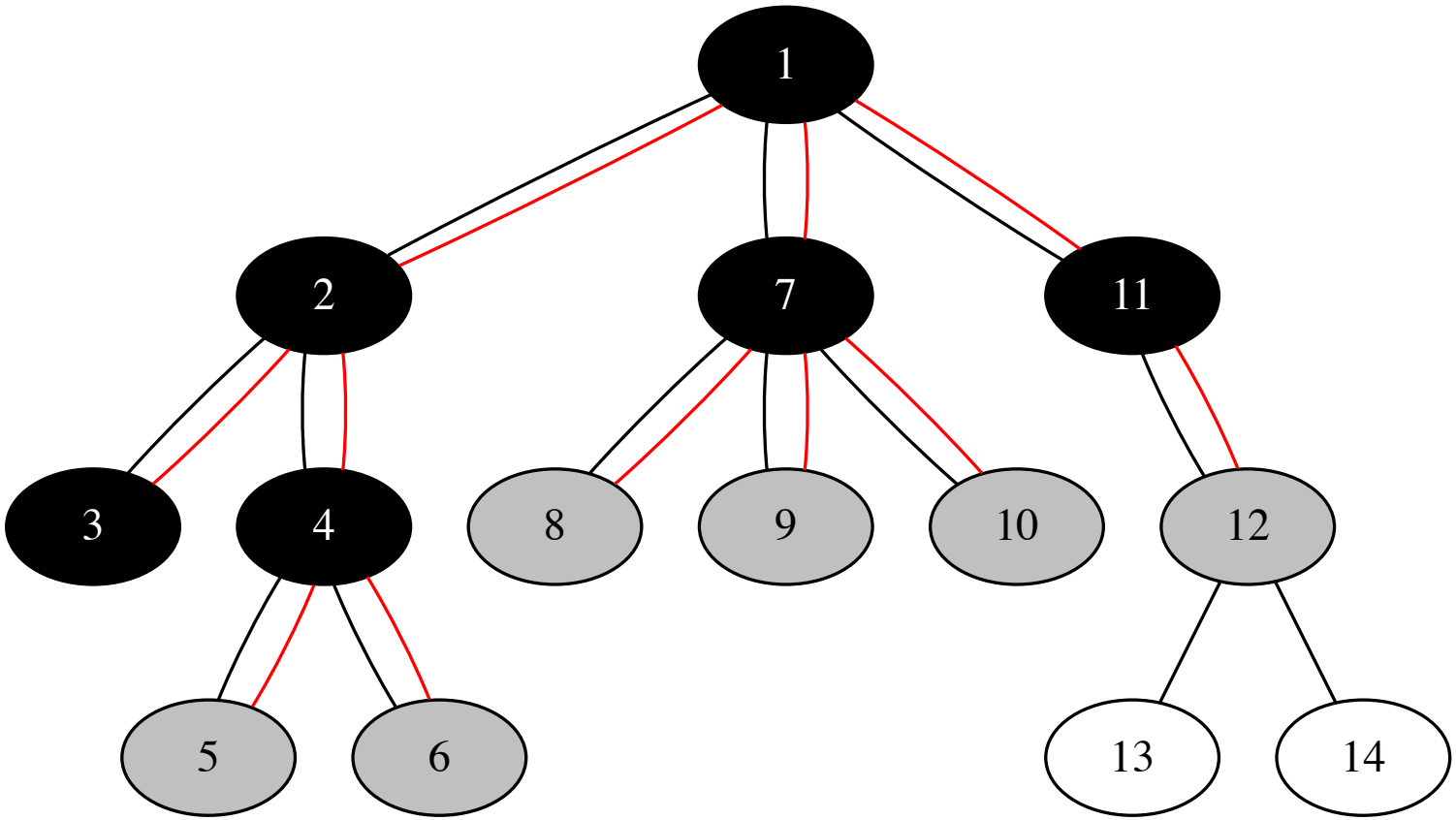
Parcours en largeur sur un arbre



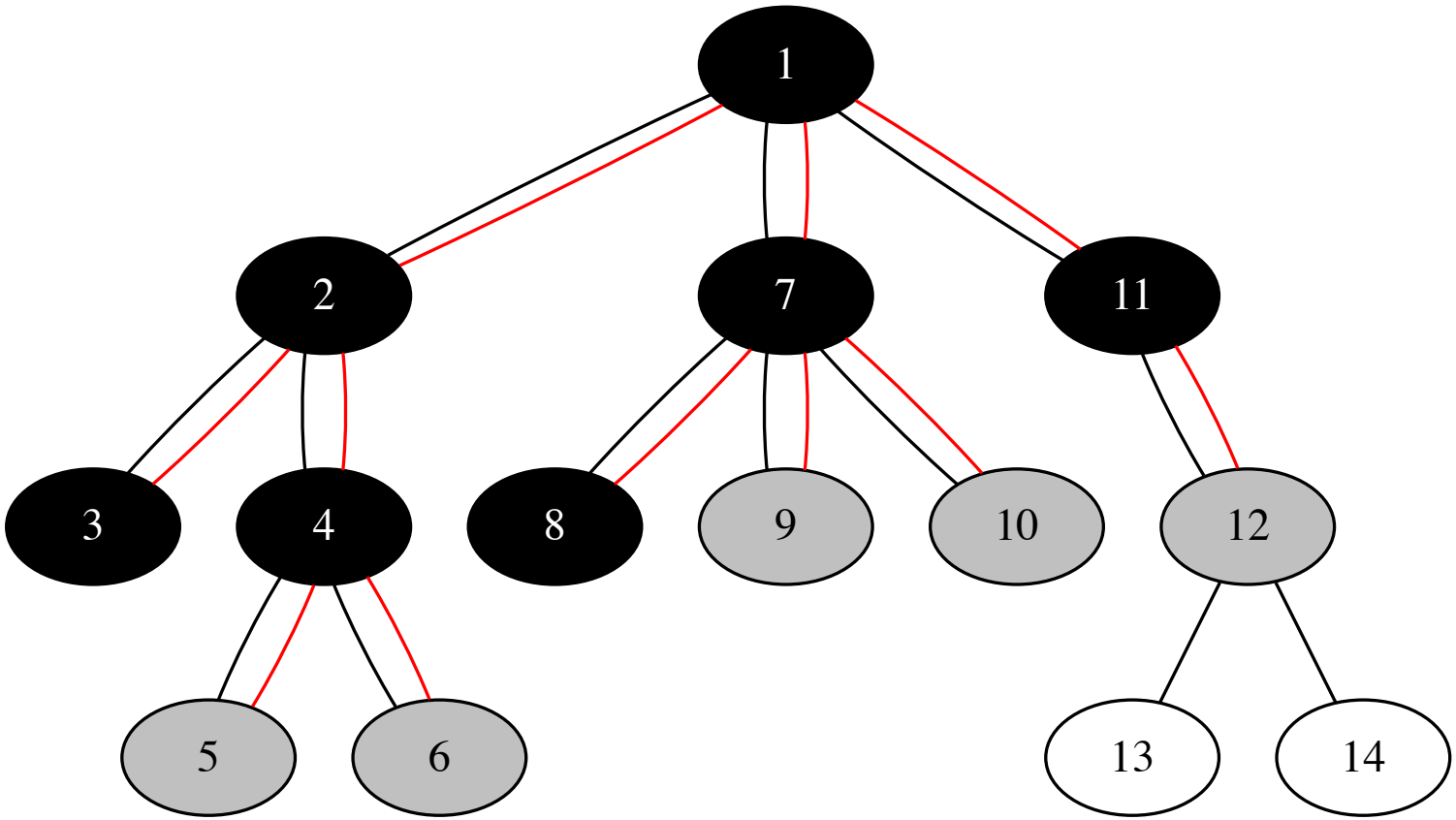
Parcours en largeur sur un arbre



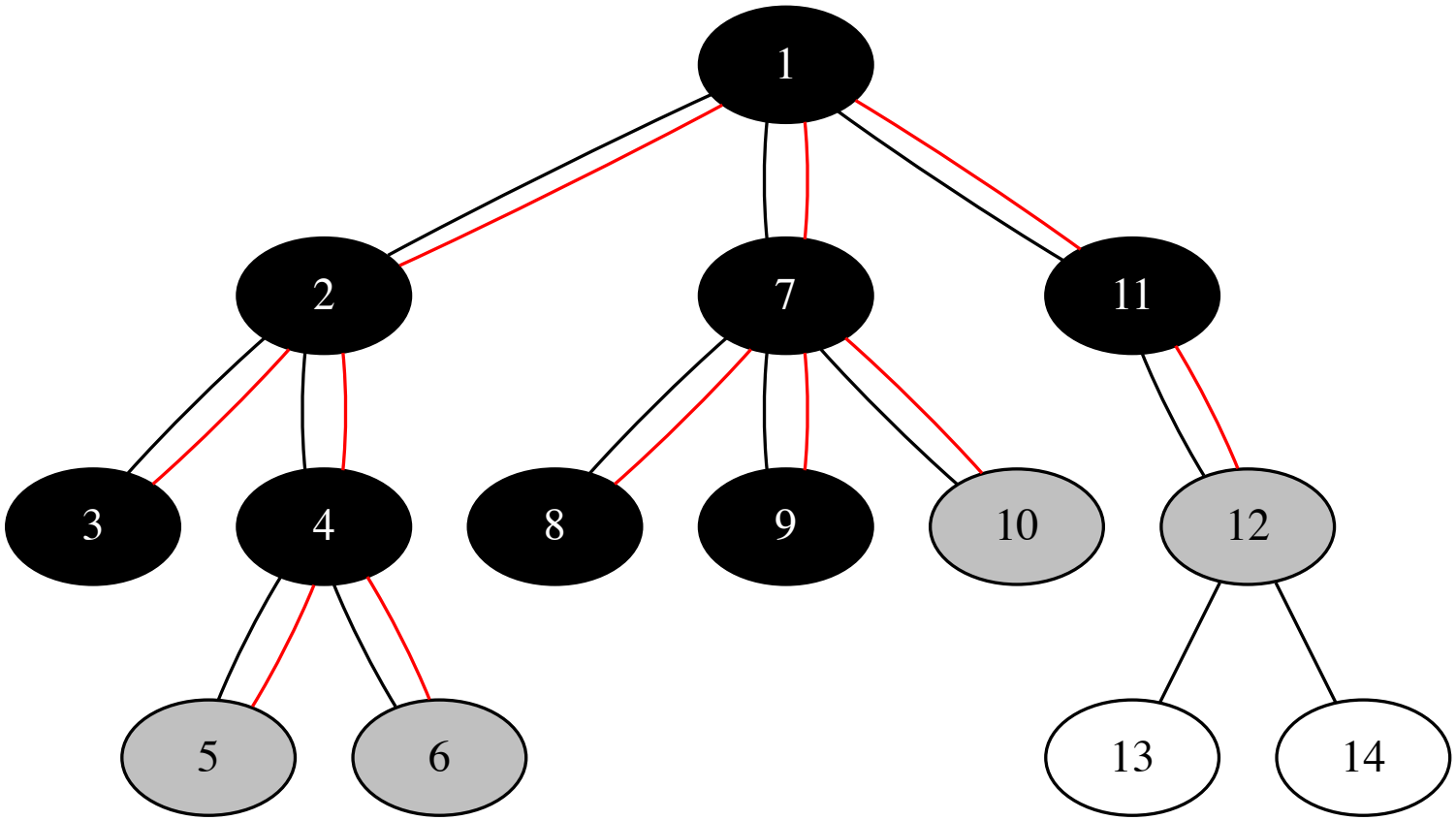
Parcours en largeur sur un arbre



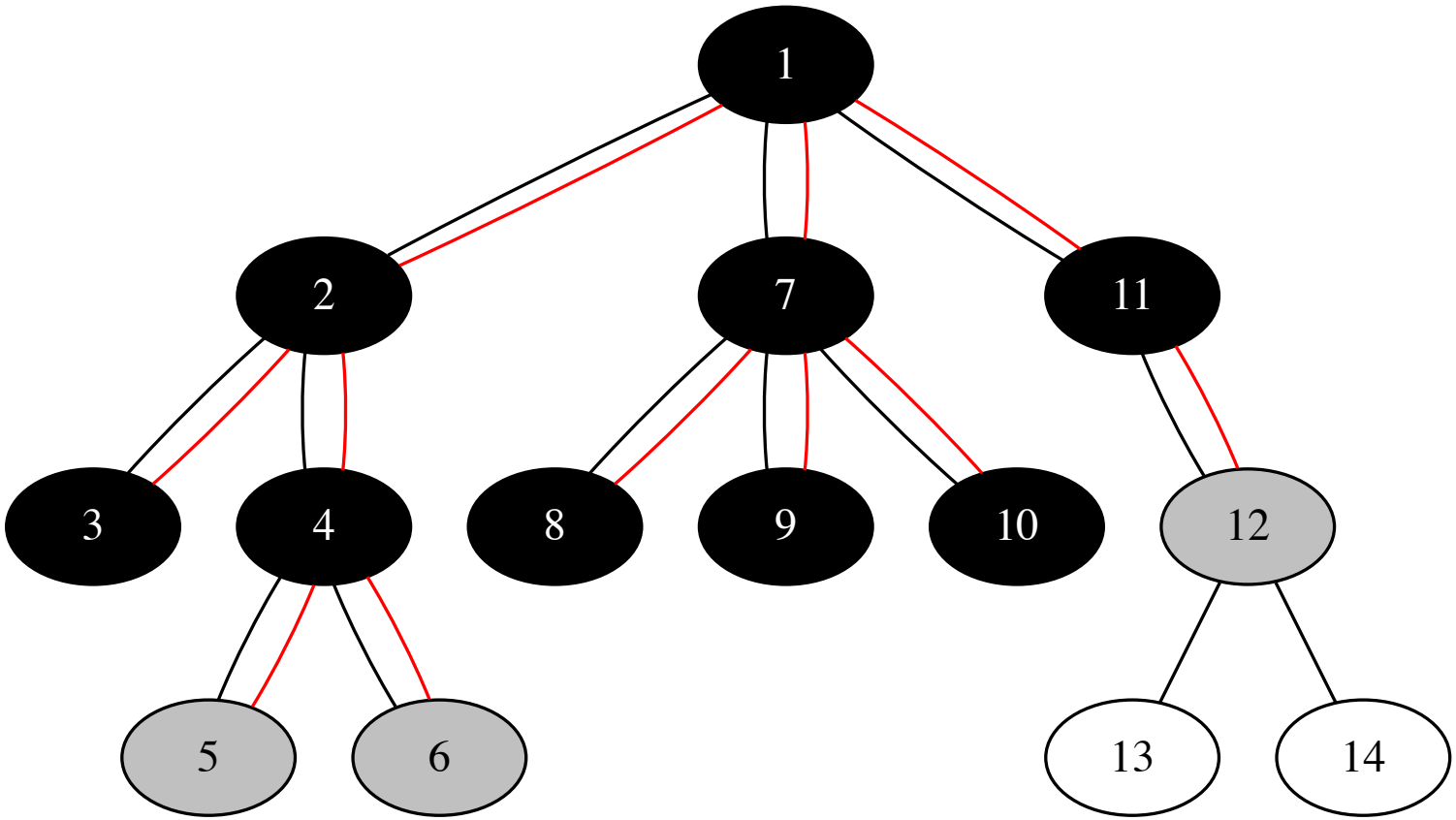
Parcours en largeur sur un arbre



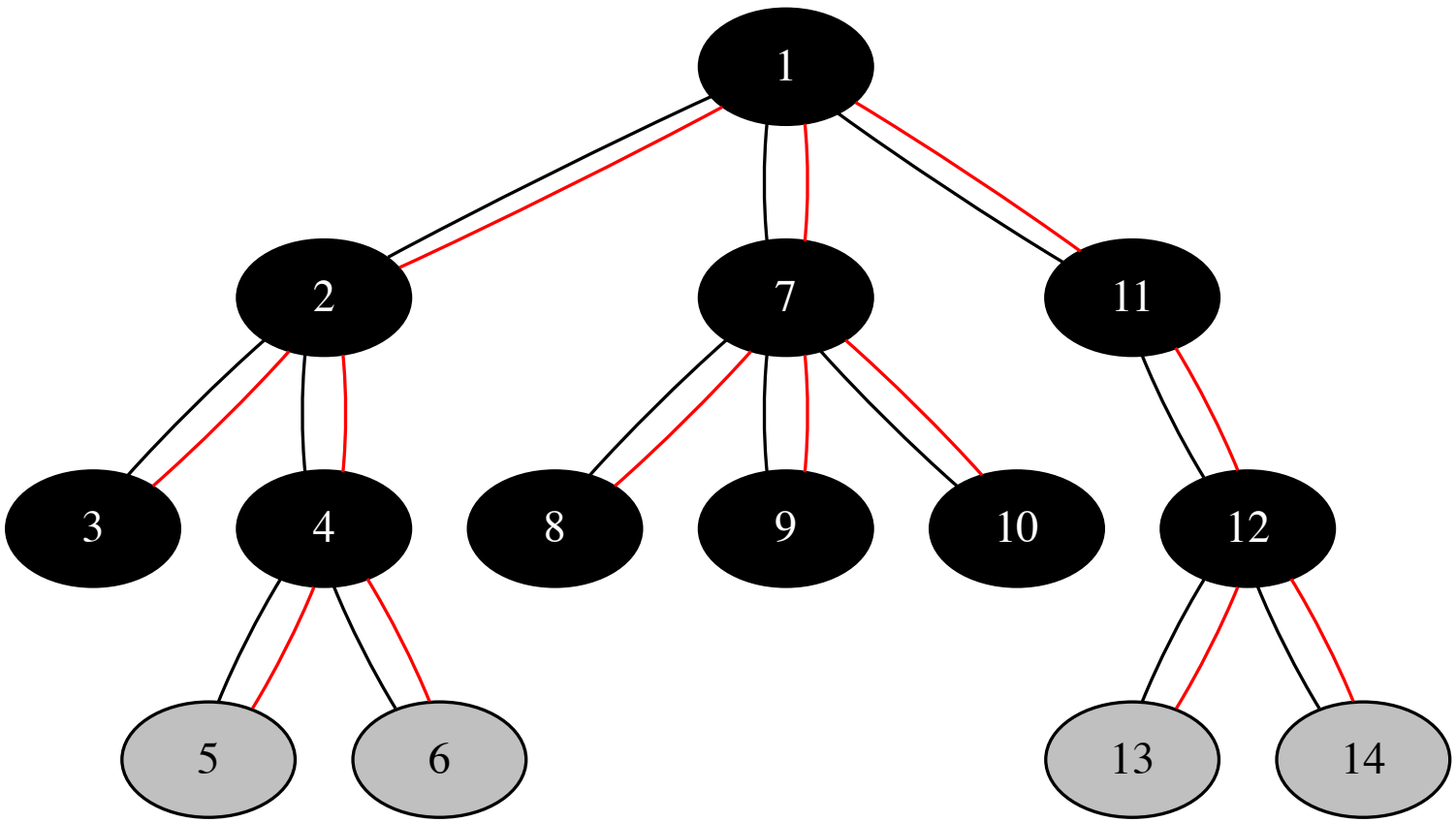
Parcours en largeur sur un arbre



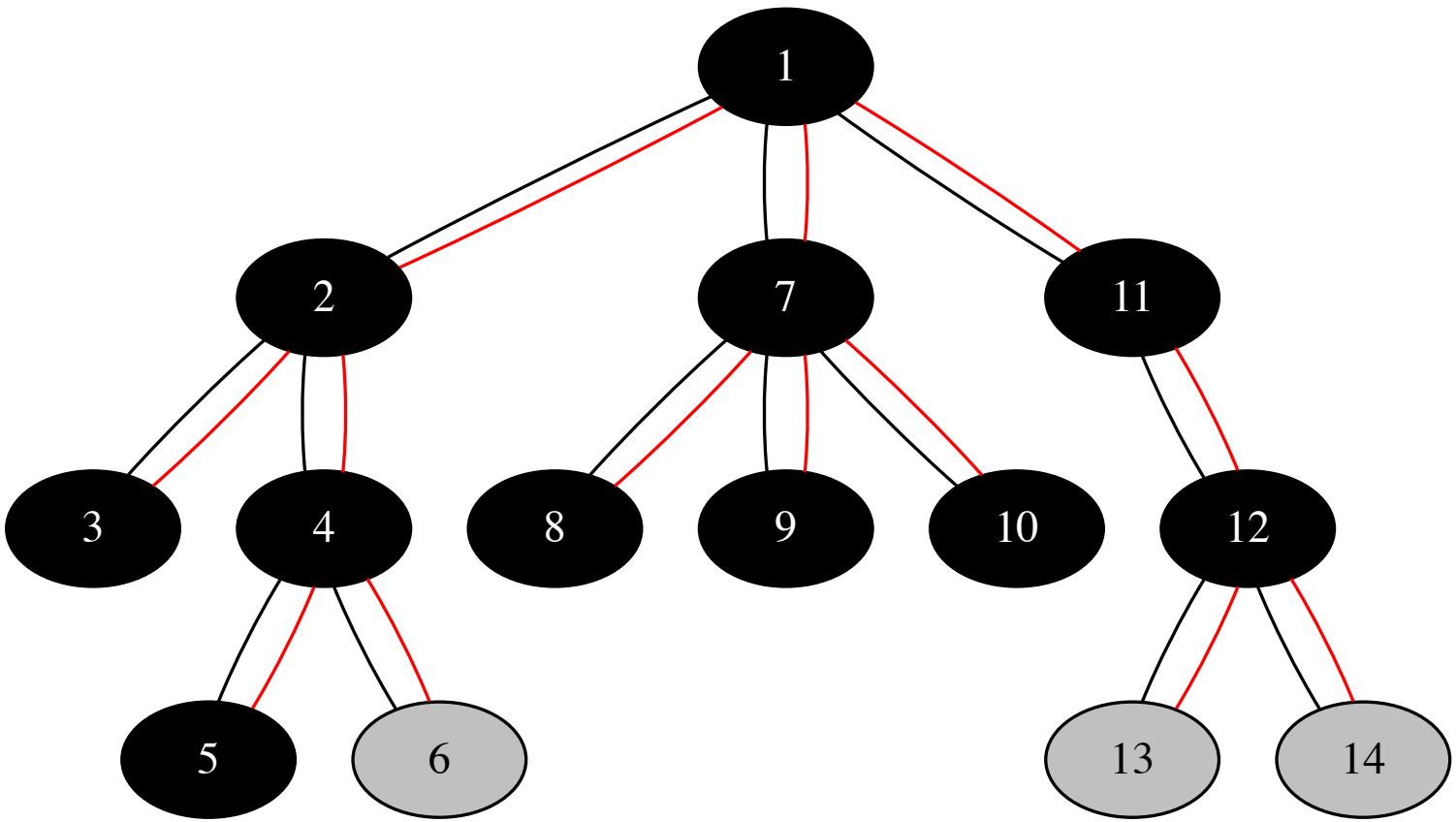
Parcours en largeur sur un arbre



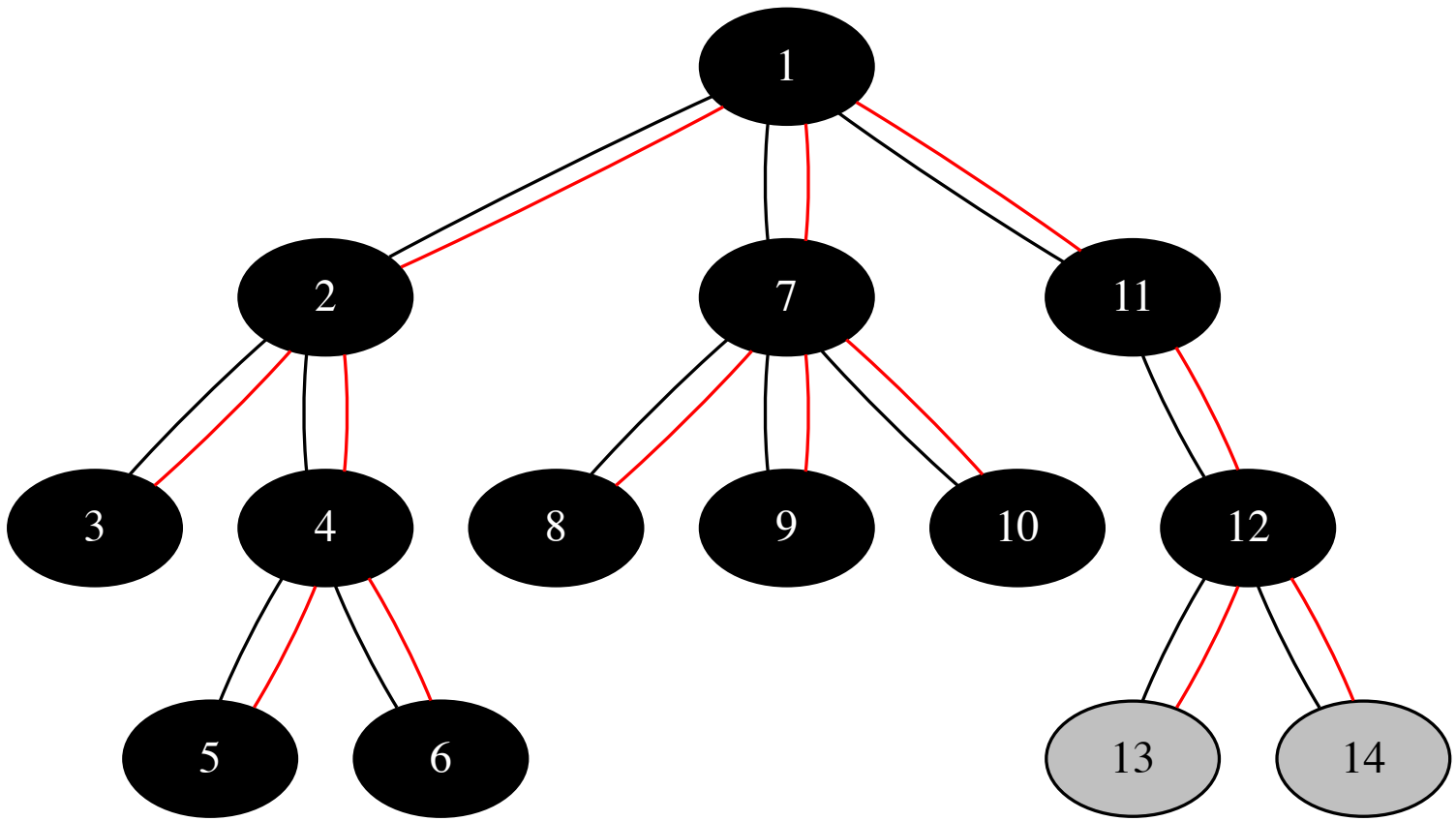
Parcours en largeur sur un arbre



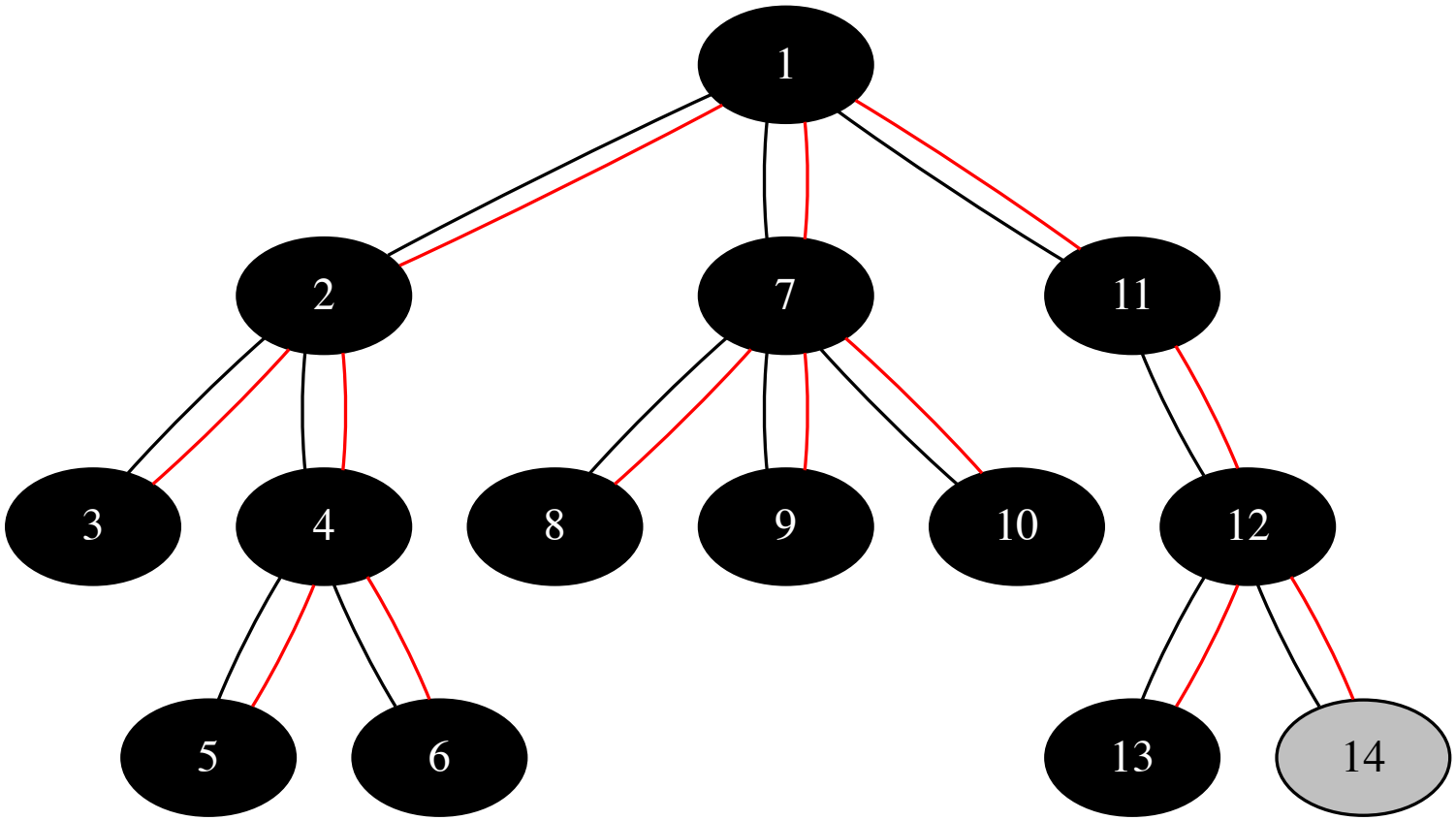
Parcours en largeur sur un arbre



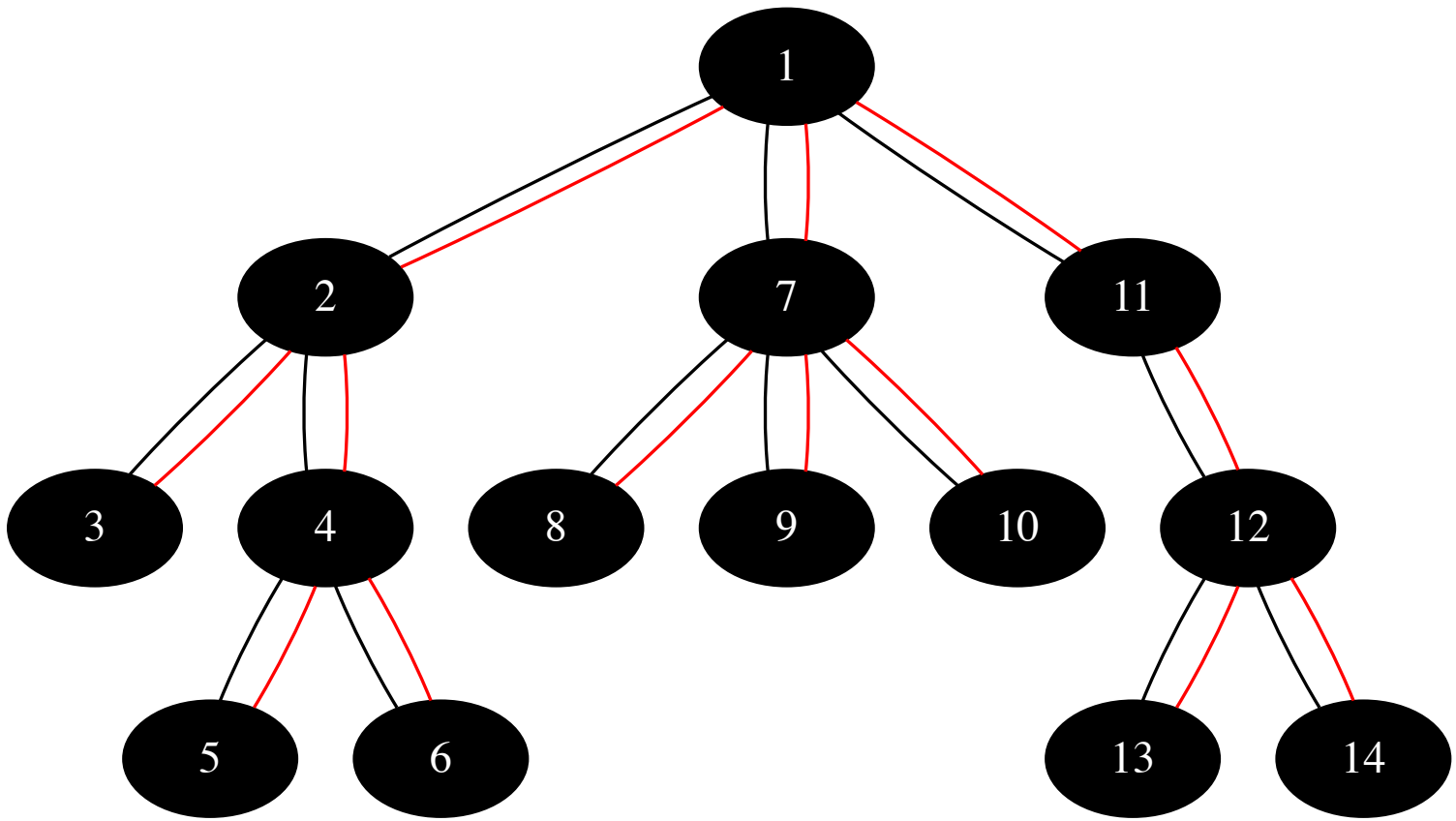
Parcours en largeur sur un arbre



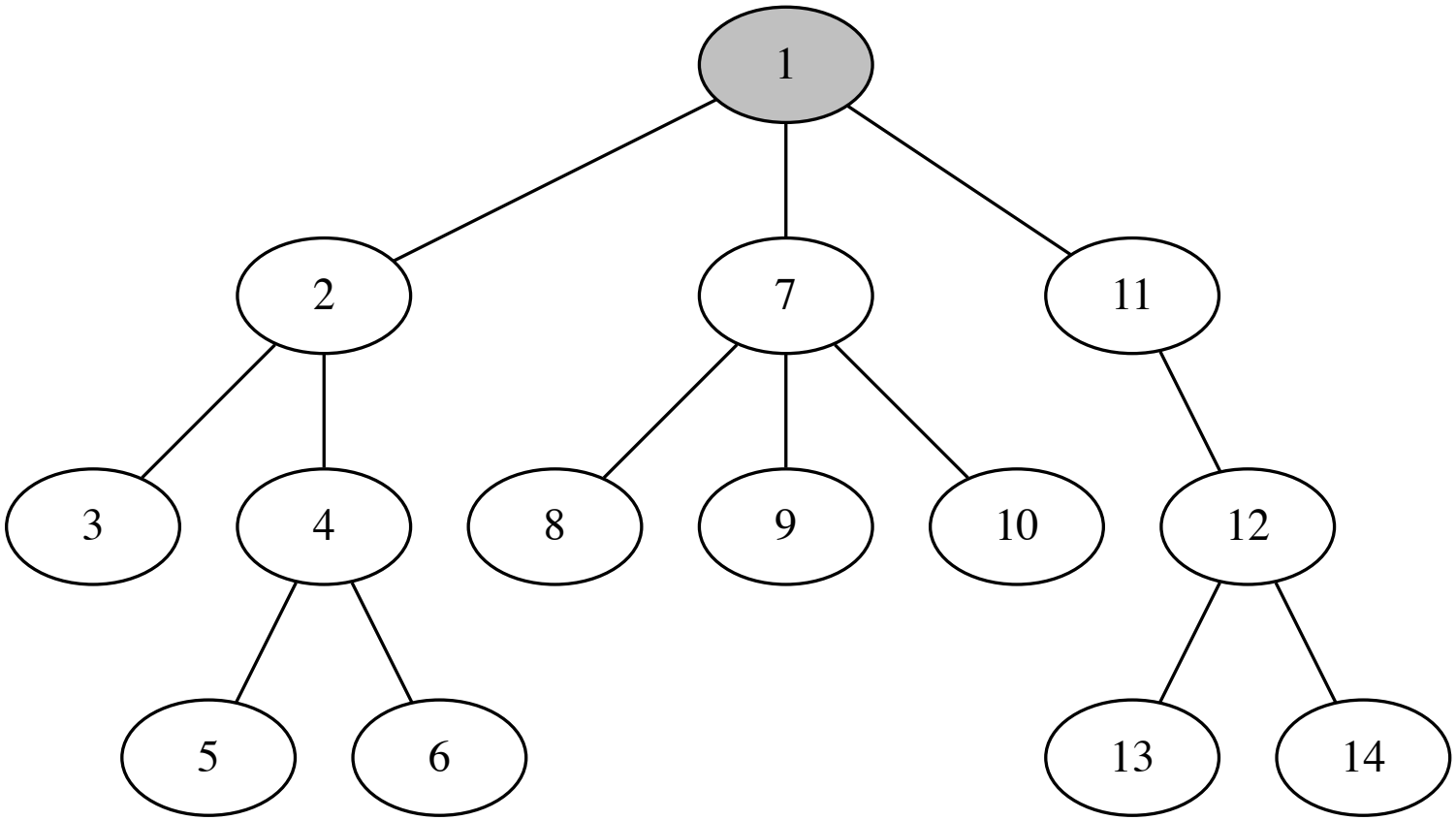
Parcours en largeur sur un arbre



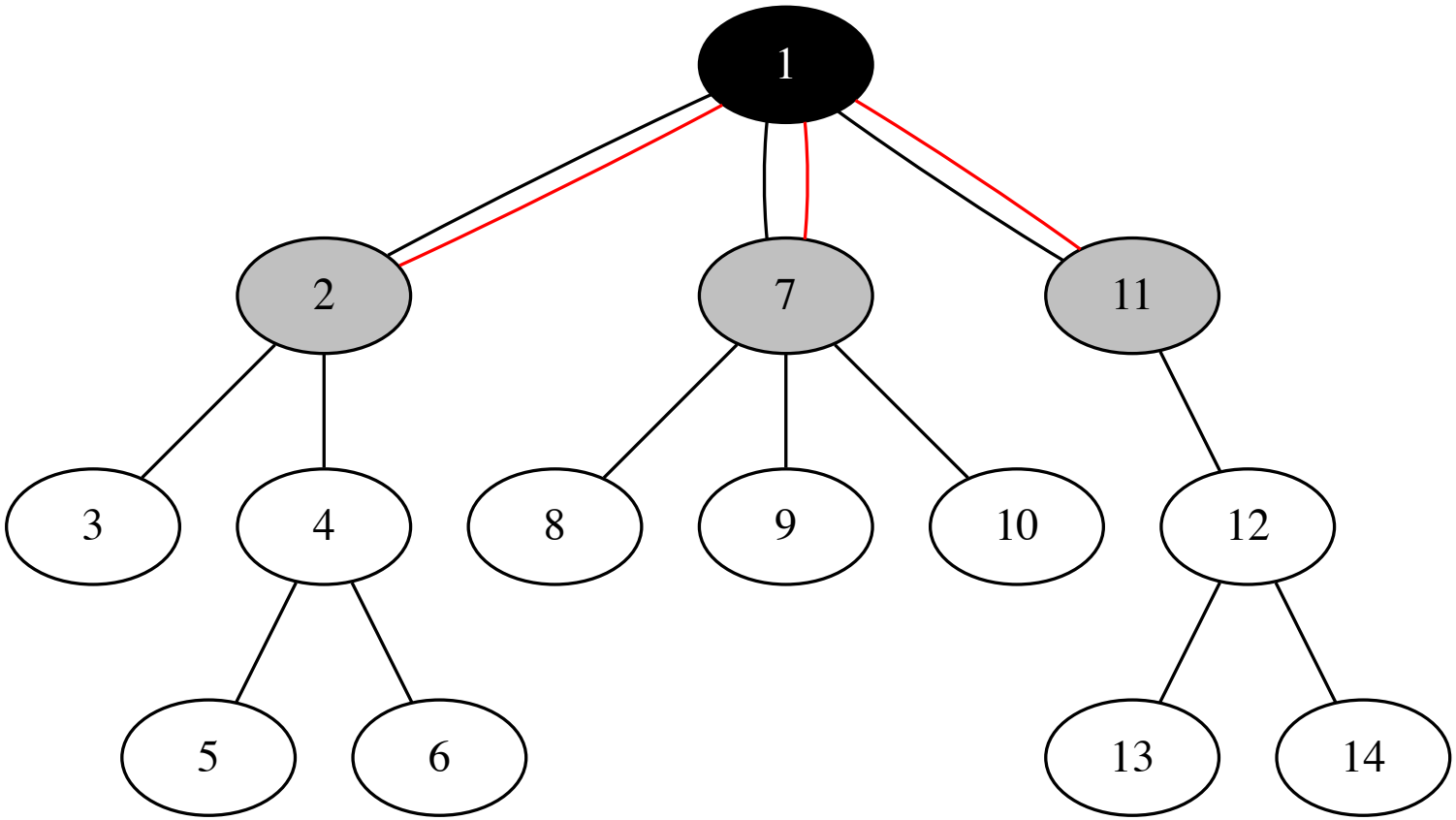
Parcours en largeur sur un arbre



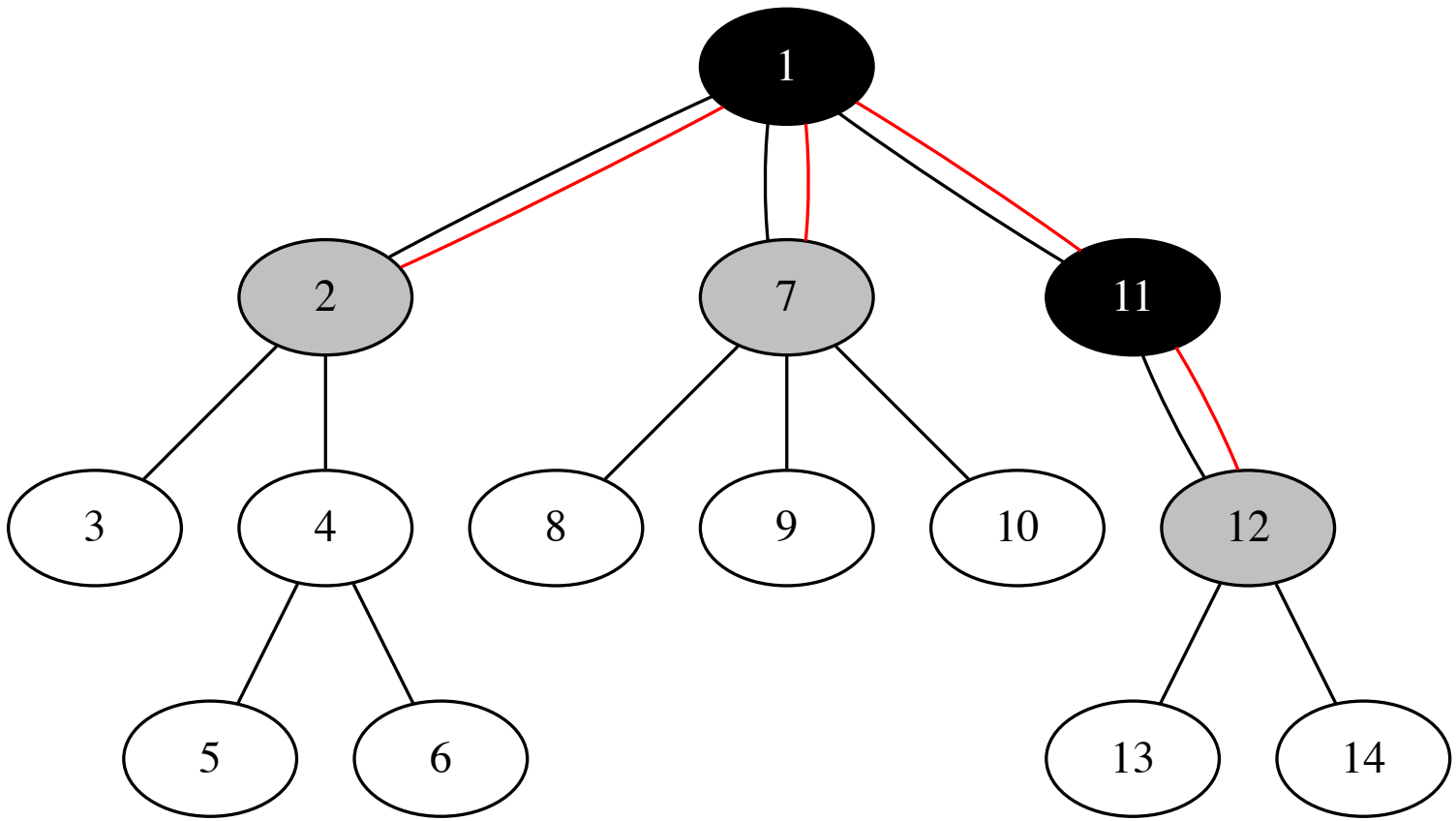
Parcours en profondeur sur un arbre



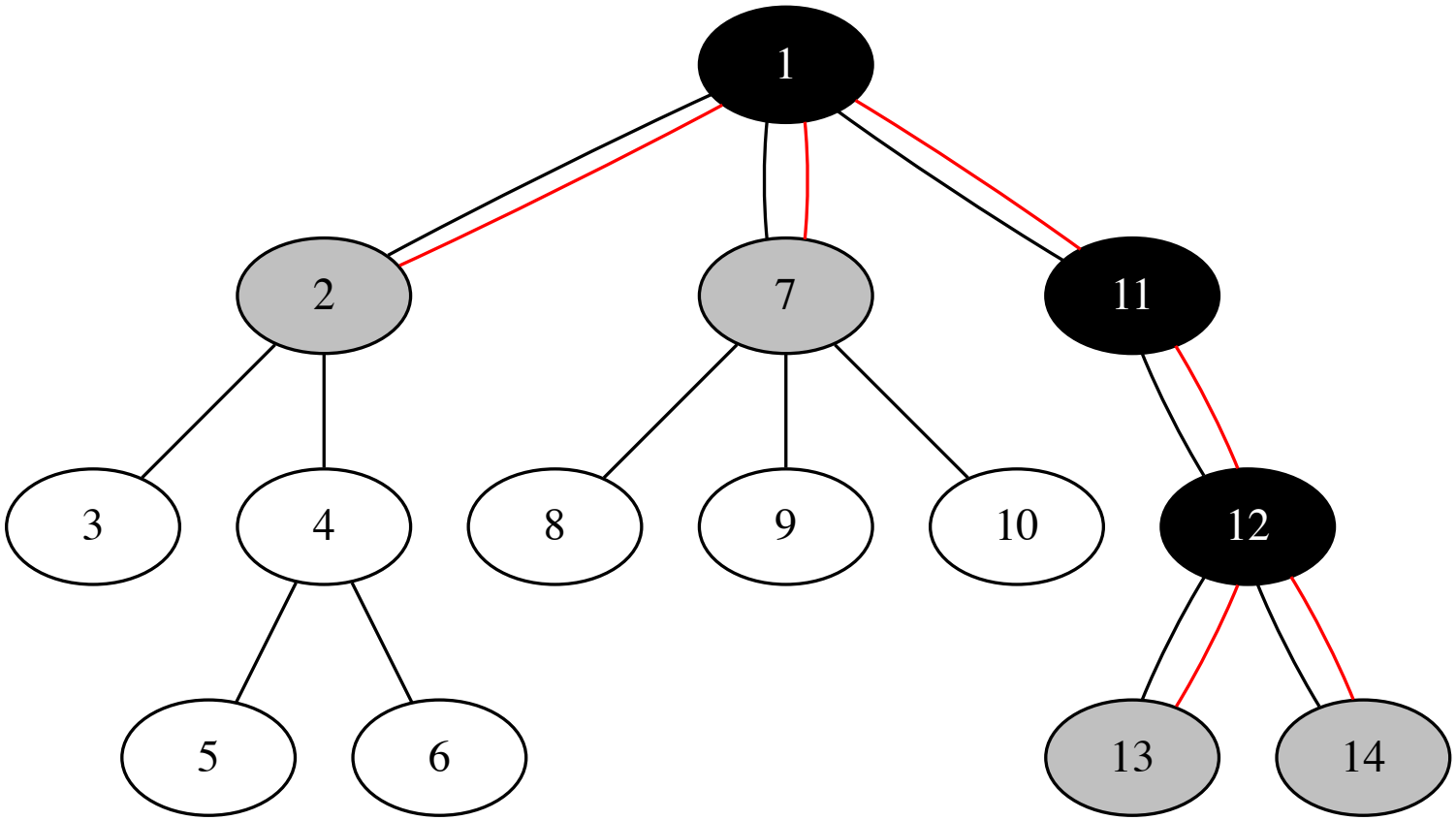
Parcours en profondeur sur un arbre



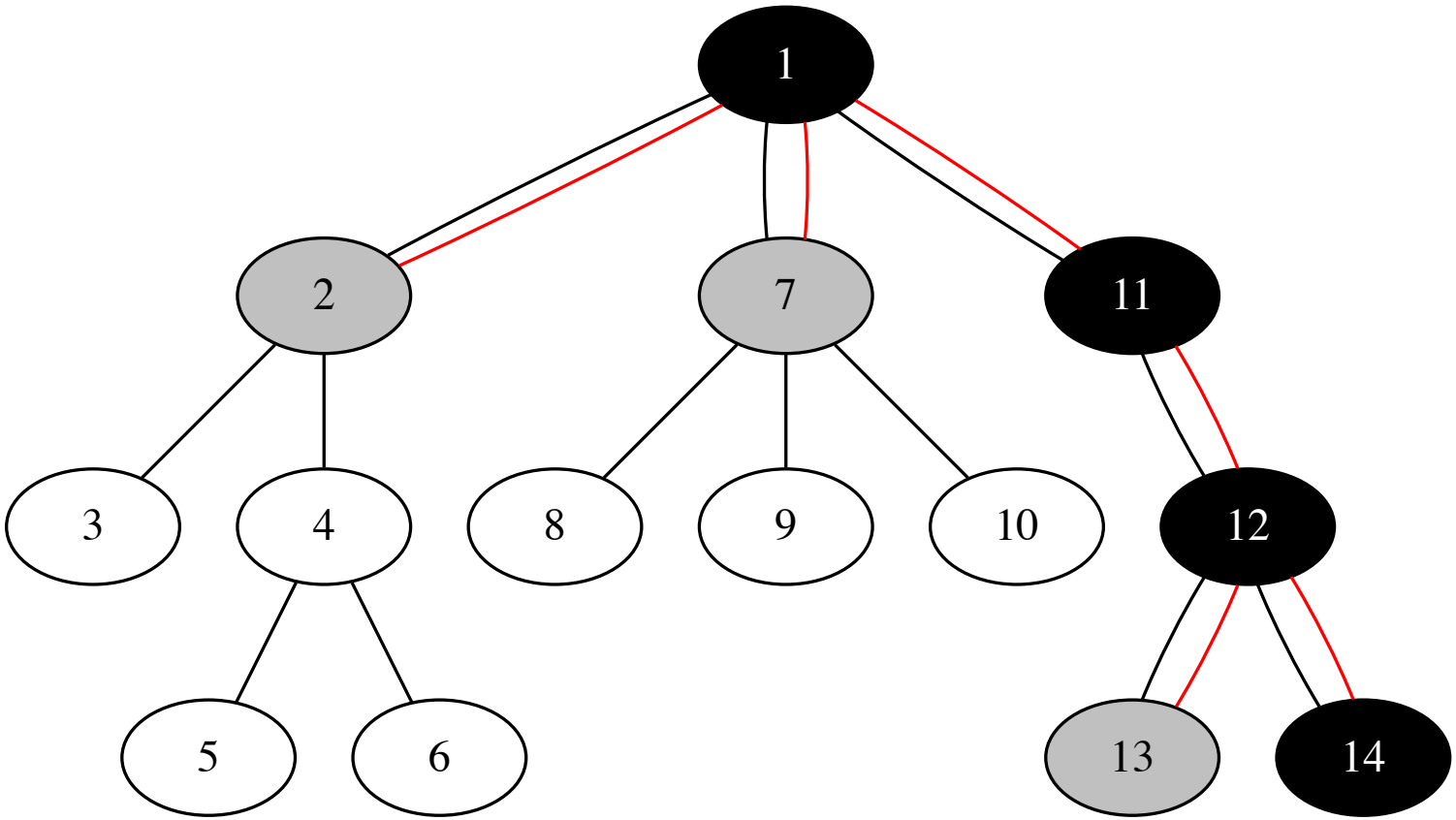
Parcours en profondeur sur un arbre



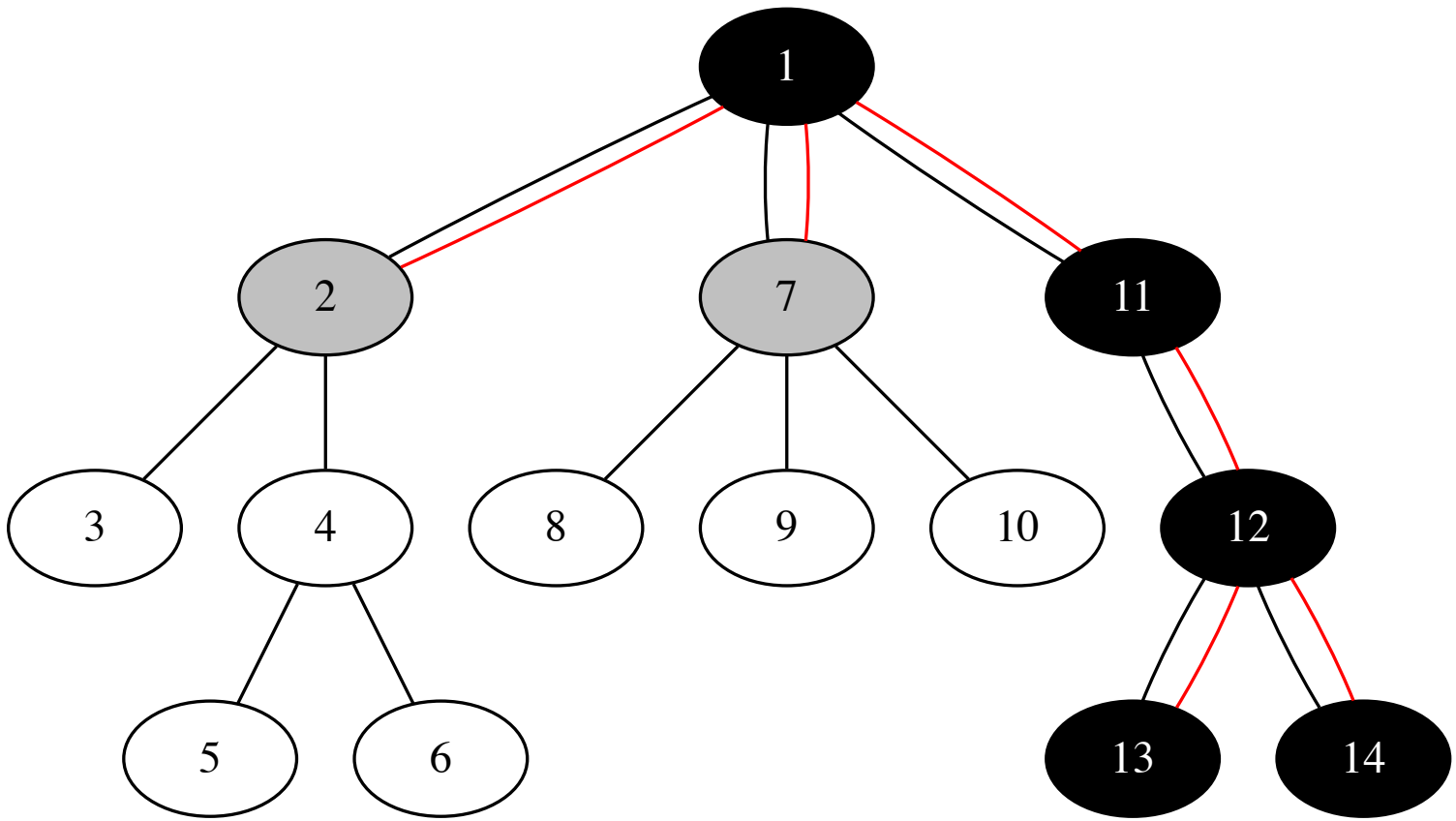
Parcours en profondeur sur un arbre



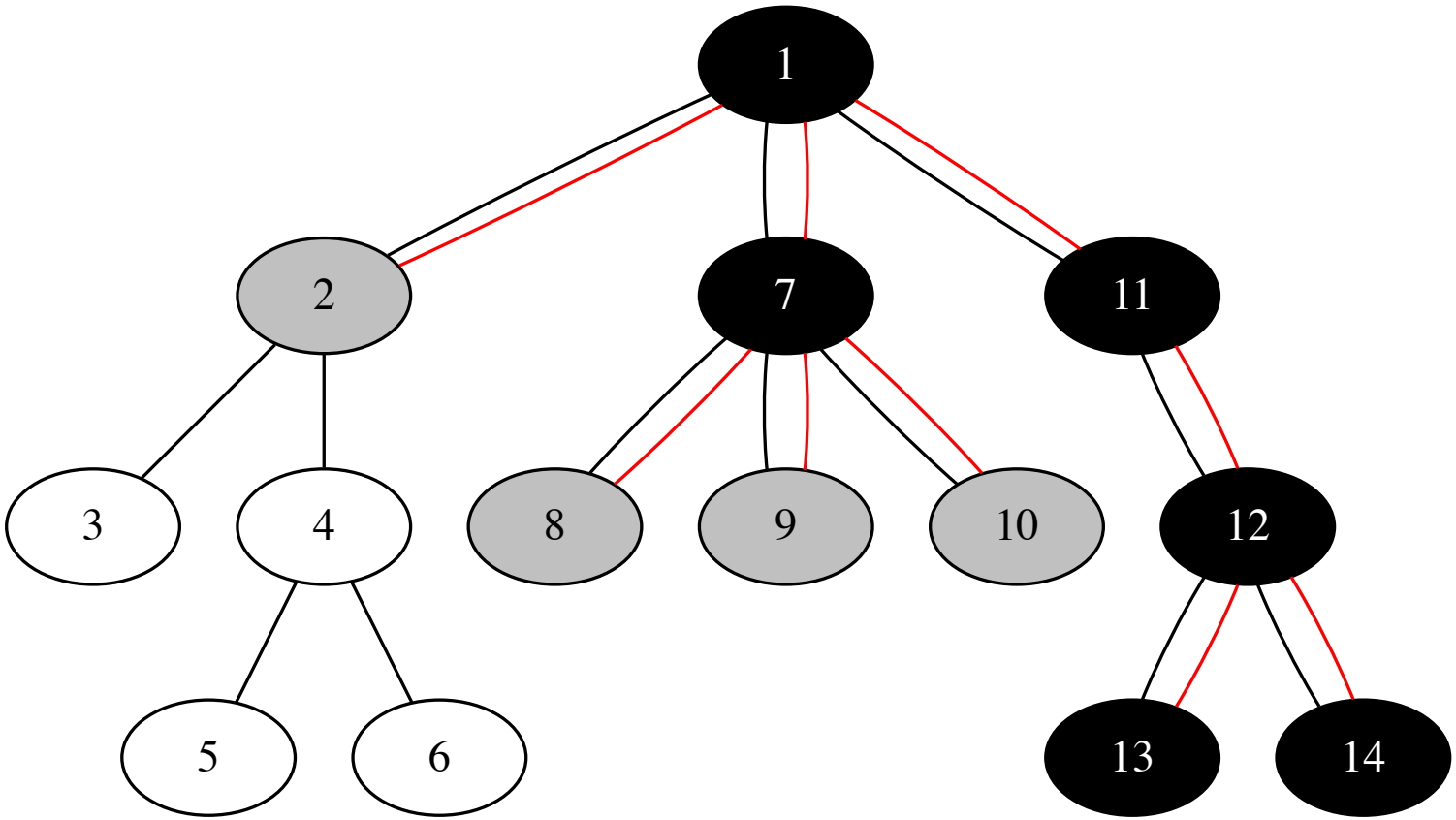
Parcours en profondeur sur un arbre



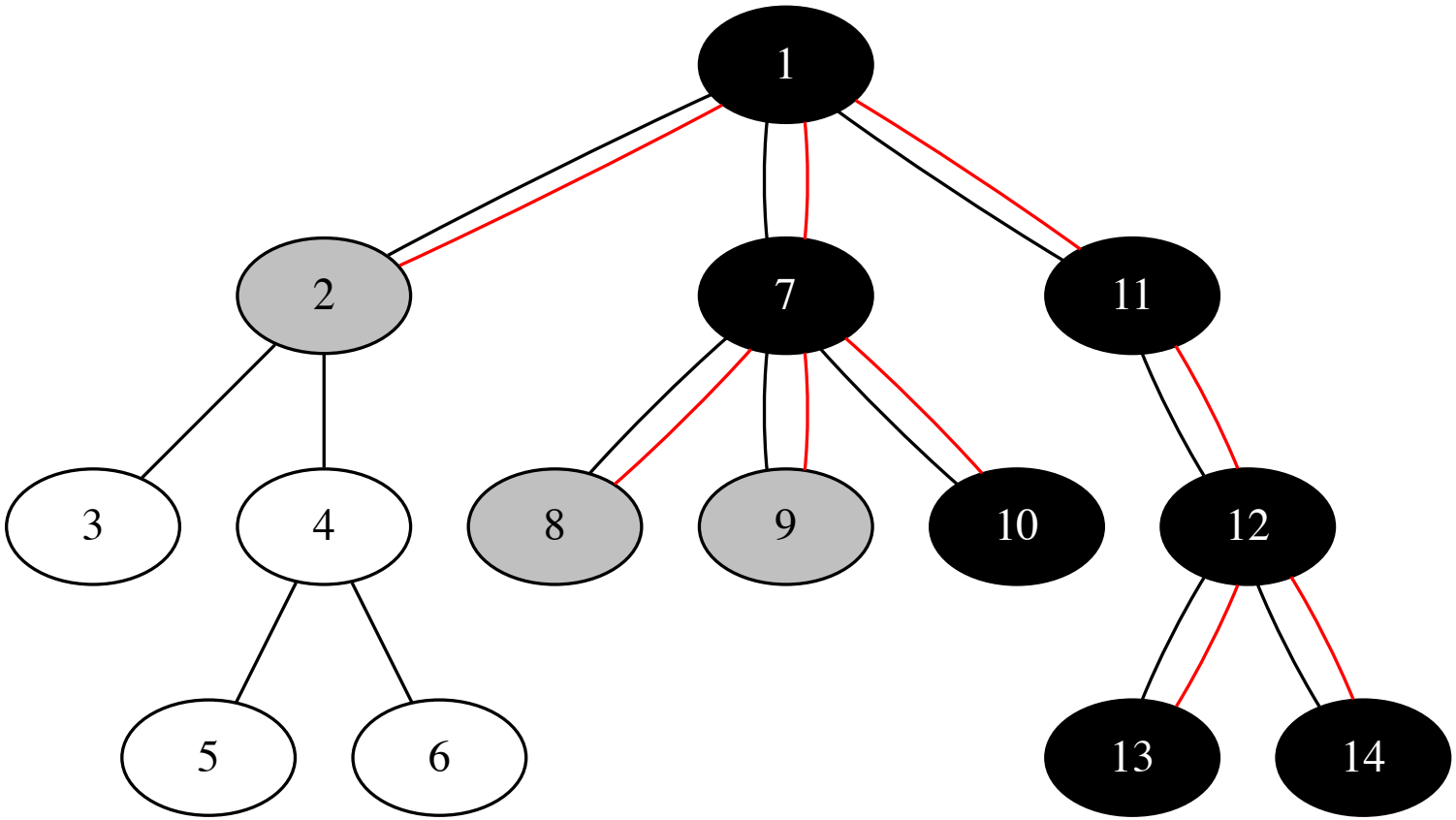
Parcours en profondeur sur un arbre



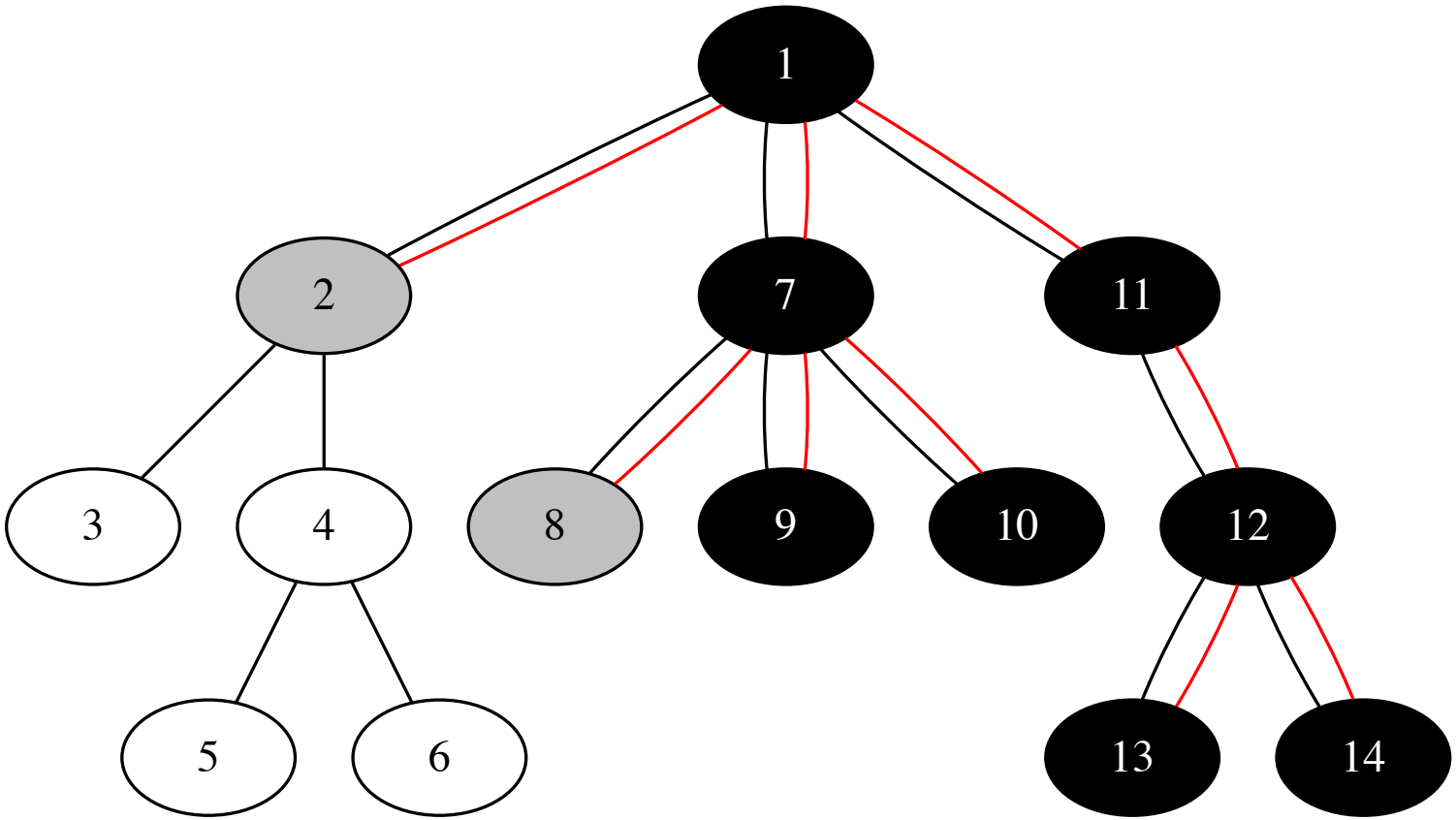
Parcours en profondeur sur un arbre



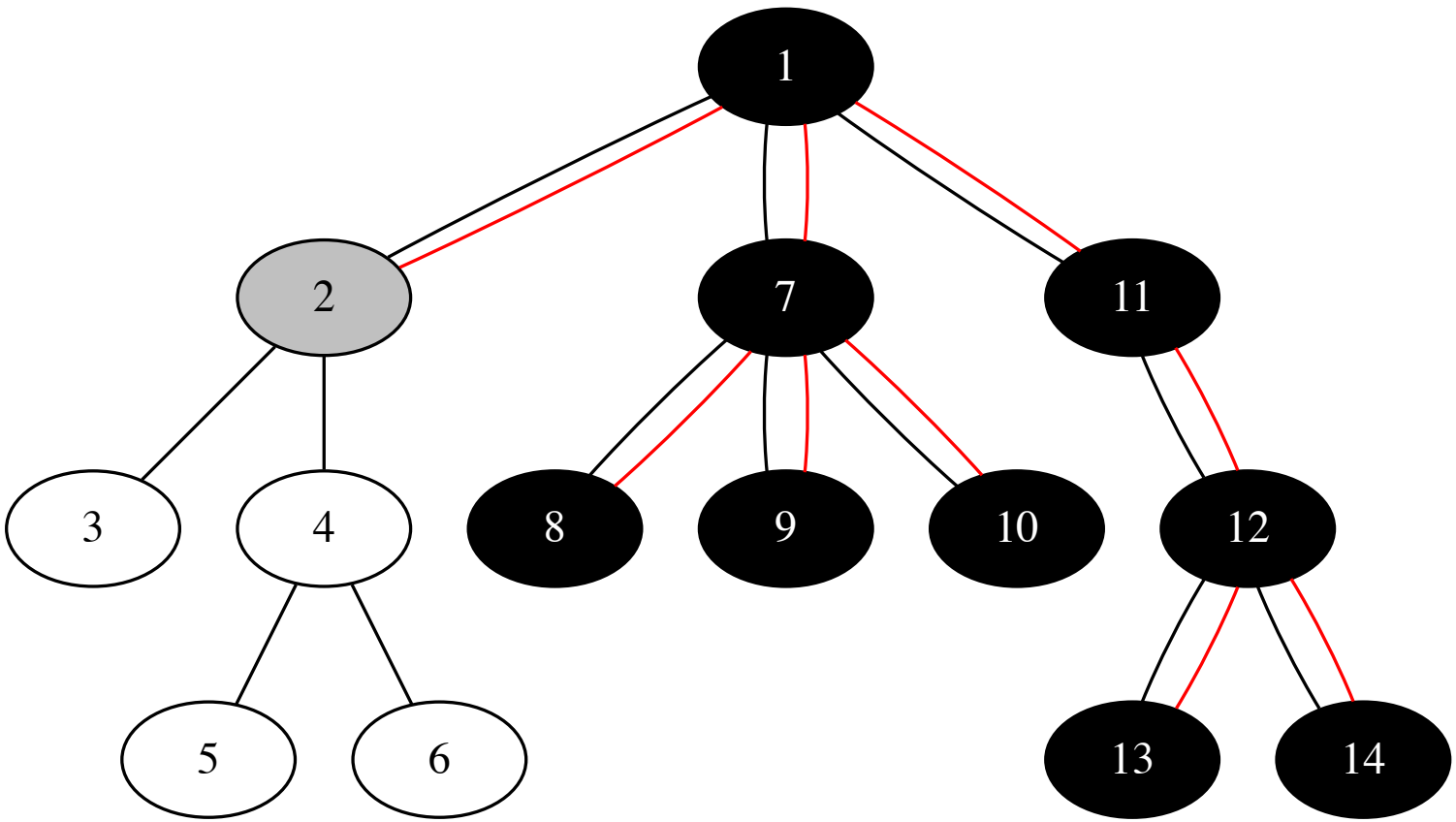
Parcours en profondeur sur un arbre



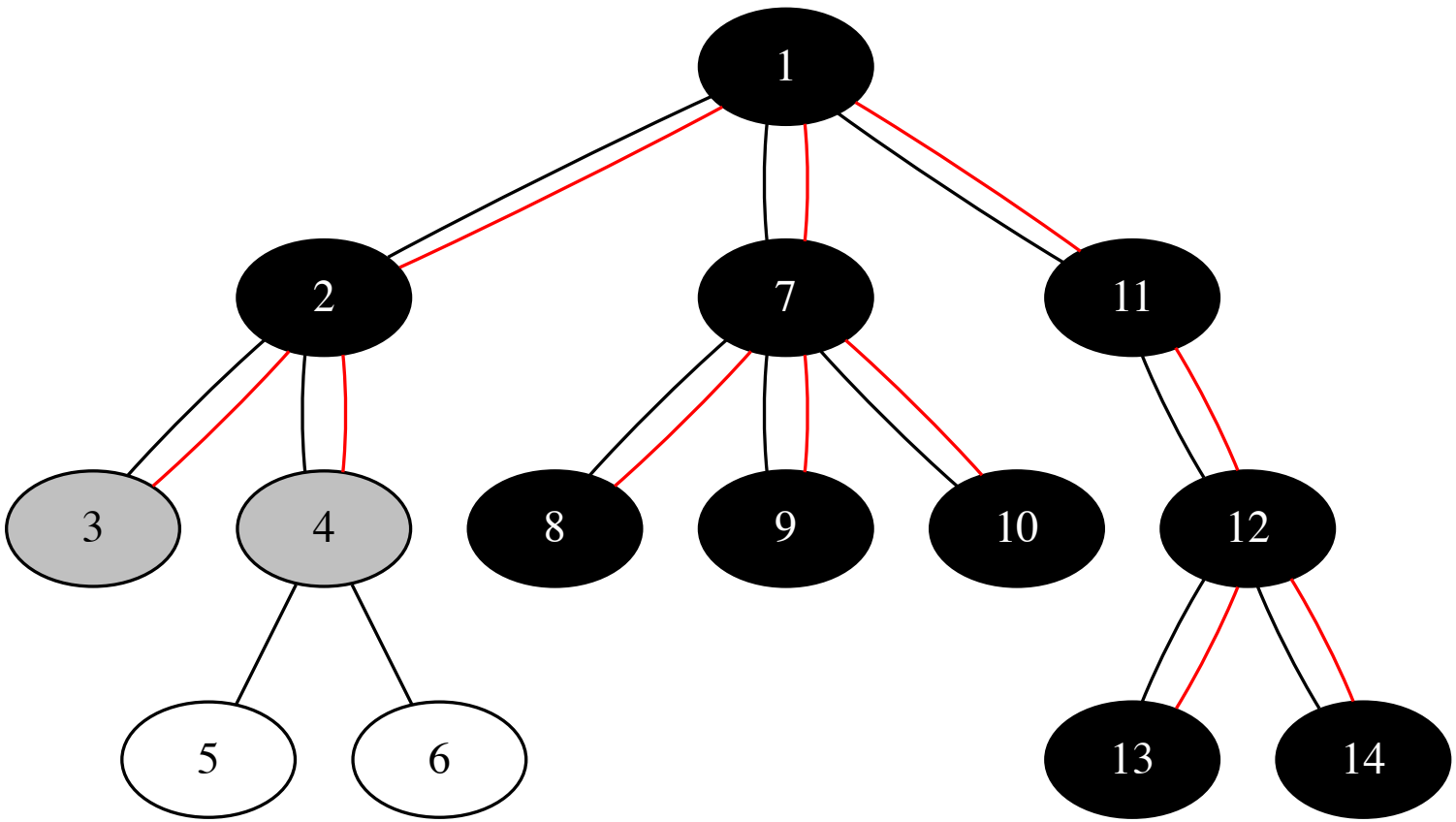
Parcours en profondeur sur un arbre



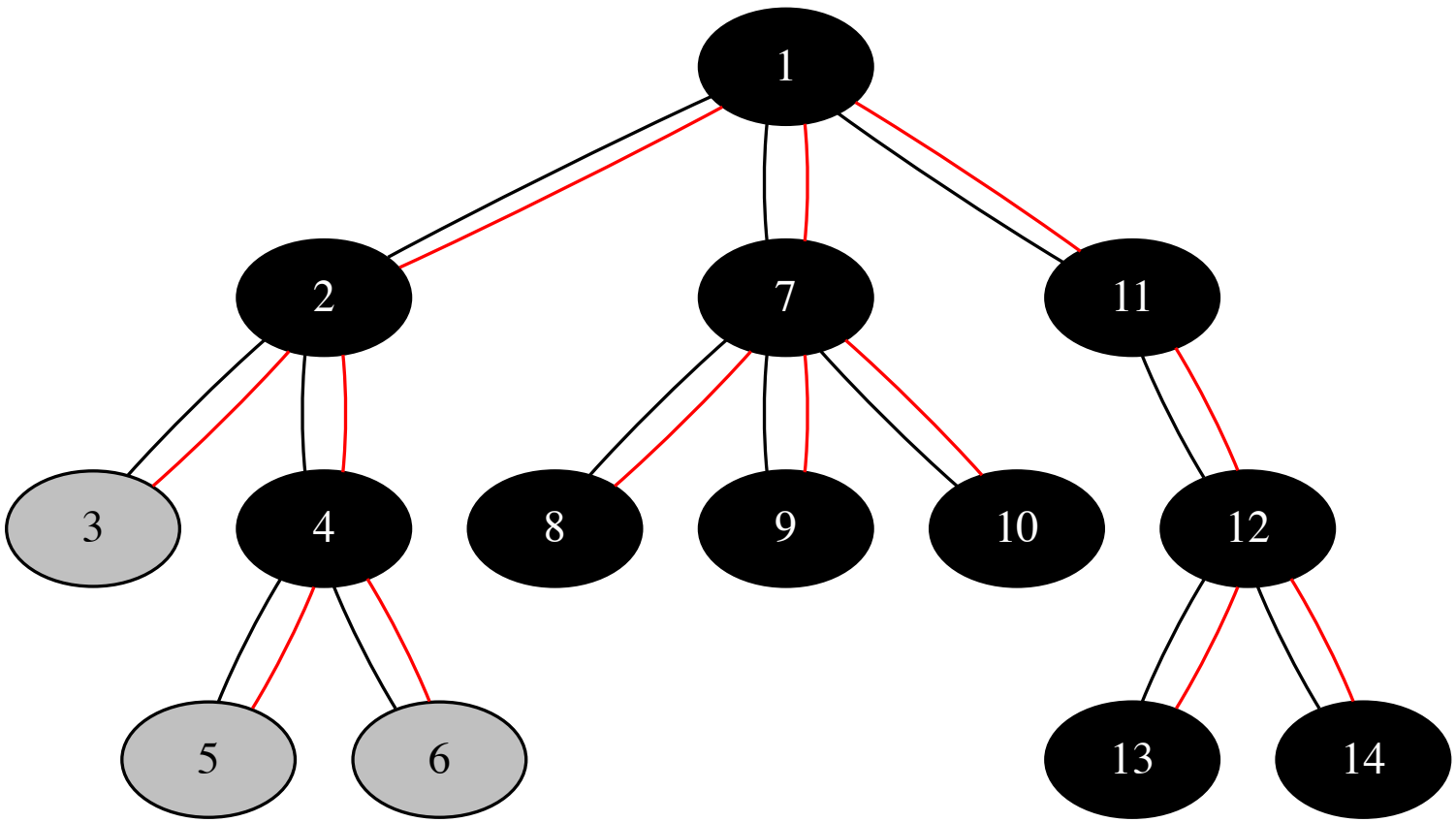
Parcours en profondeur sur un arbre



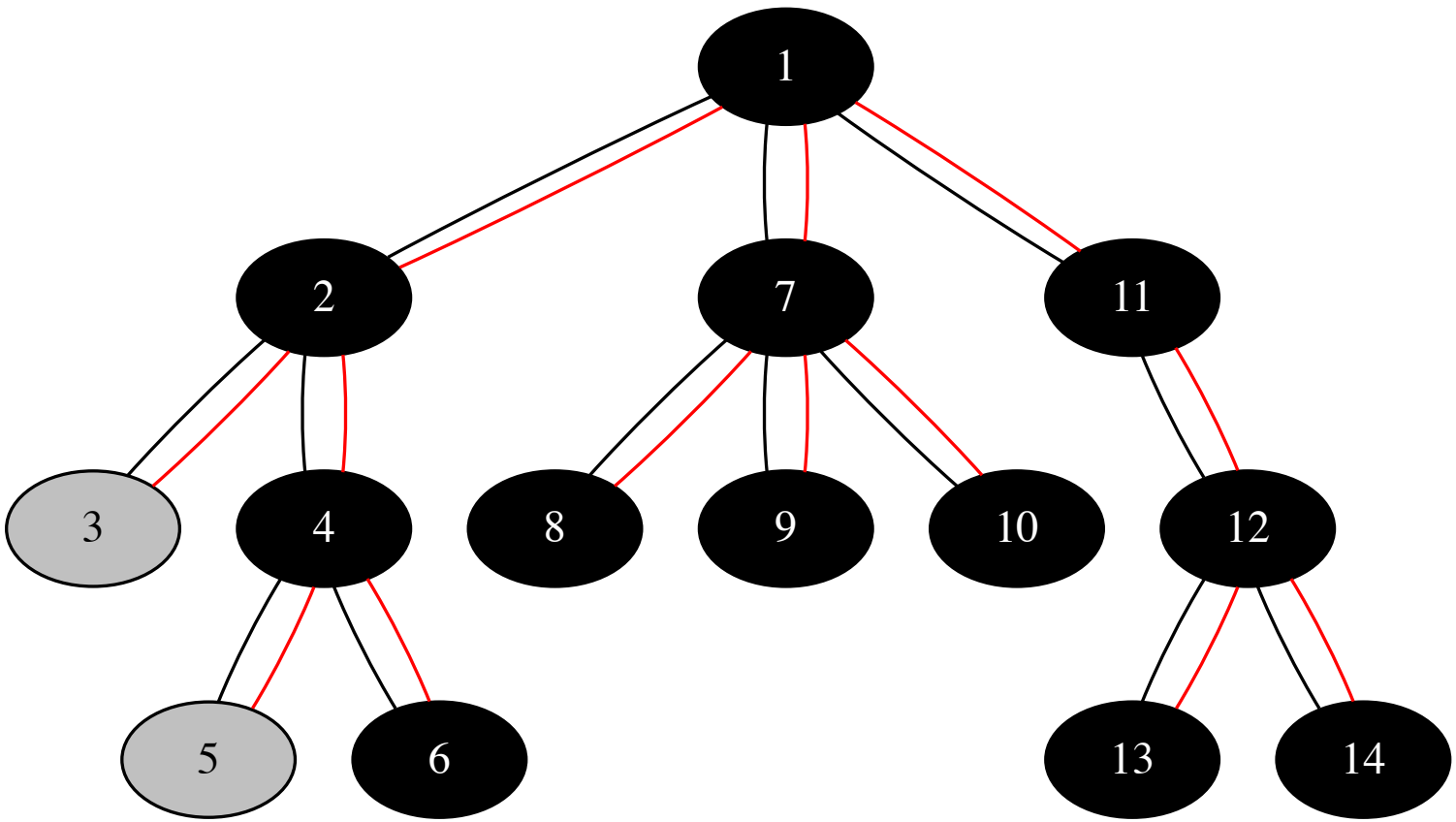
Parcours en profondeur sur un arbre



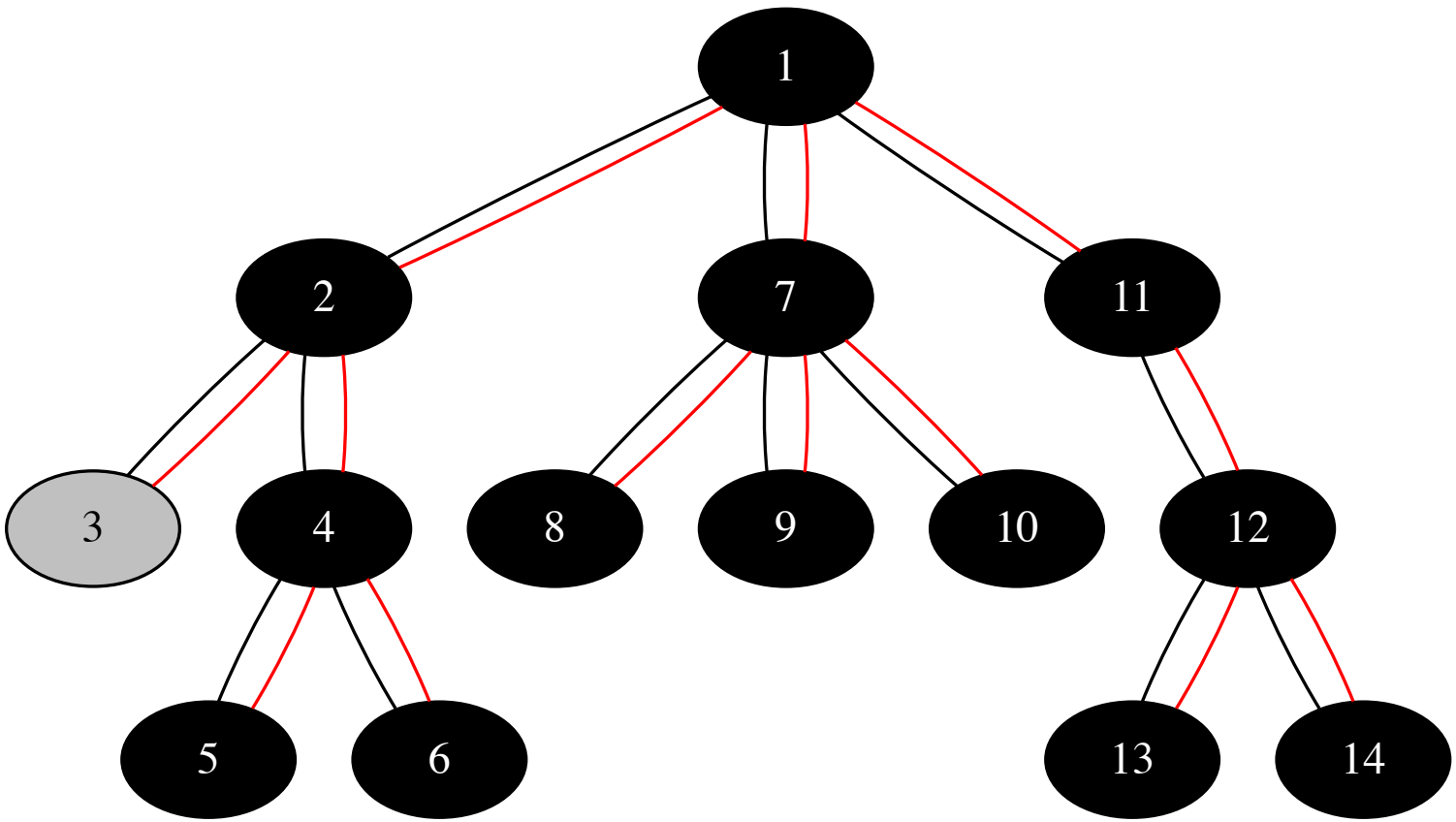
Parcours en profondeur sur un arbre



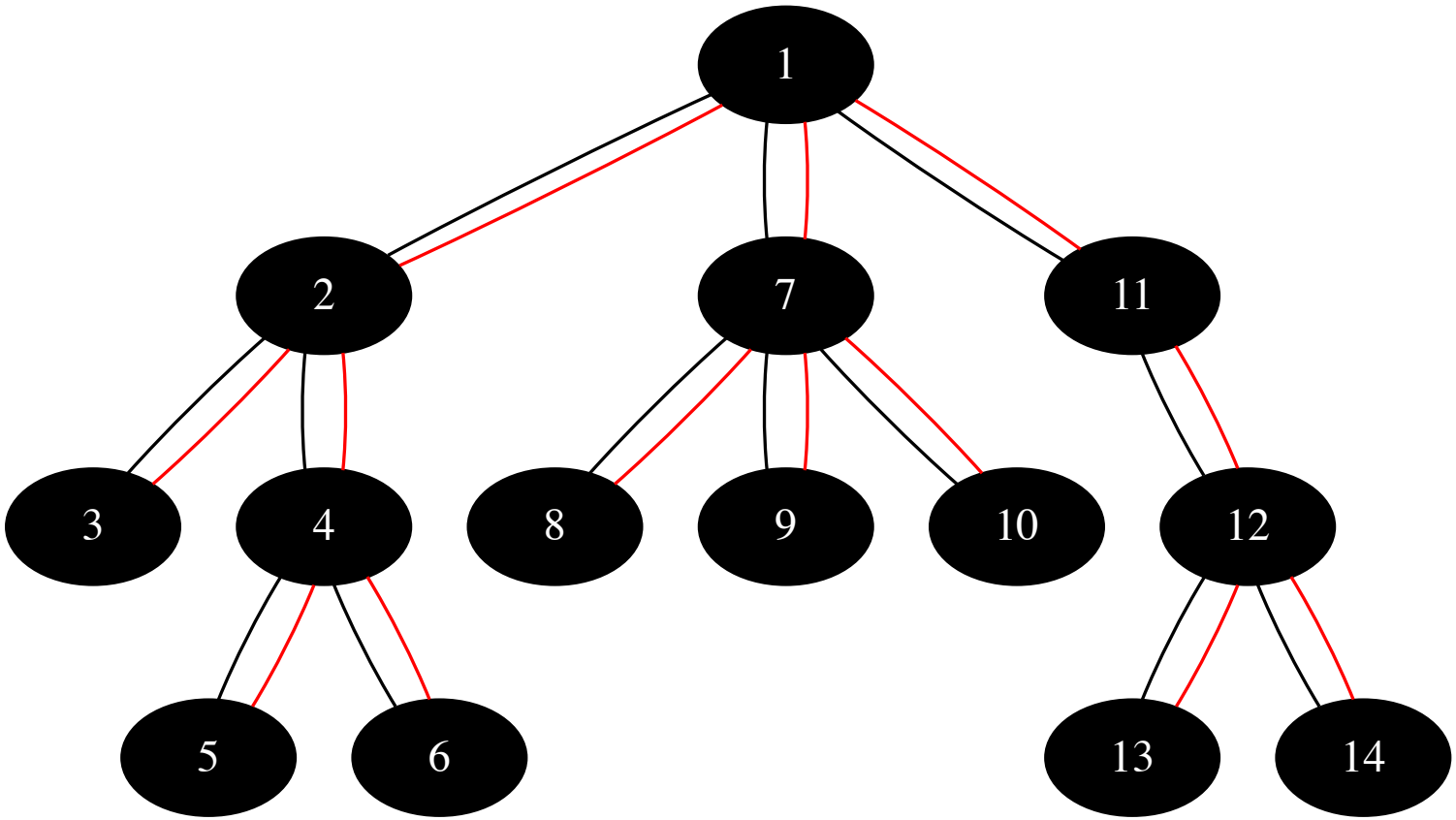
Parcours en profondeur sur un arbre



Parcours en profondeur sur un arbre



Parcours en profondeur sur un arbre



Parcours, applications

1. Graphes :
   - Recherche d'un chemin entre deux sommets
   - Calcul des composantes connexes
   - Detection de cycles
   - Test si un graphe est biparti
2. Recherche des voisins :
   - SNCF
   - Systèmes de navigation GPS
   - Reseaux P2P (ex. BitTorrent)
   - Réseaux sociaux
   - Communication (broadcast) dans les réseaux.
   - Robots d'indexation (crawlers)
3. Ramasse-miettes (garbage collection).

Parcours, applications

Parcours en largeur :

- Calcul de distance entre deux sommets.
  Le chemin de la racine à un sommet dans un arbre recouvrant retourné par un parcours en largeur est un plus court chemin
- Exploration/construction à la volée de graphes infinis

Parcours en profondeur :

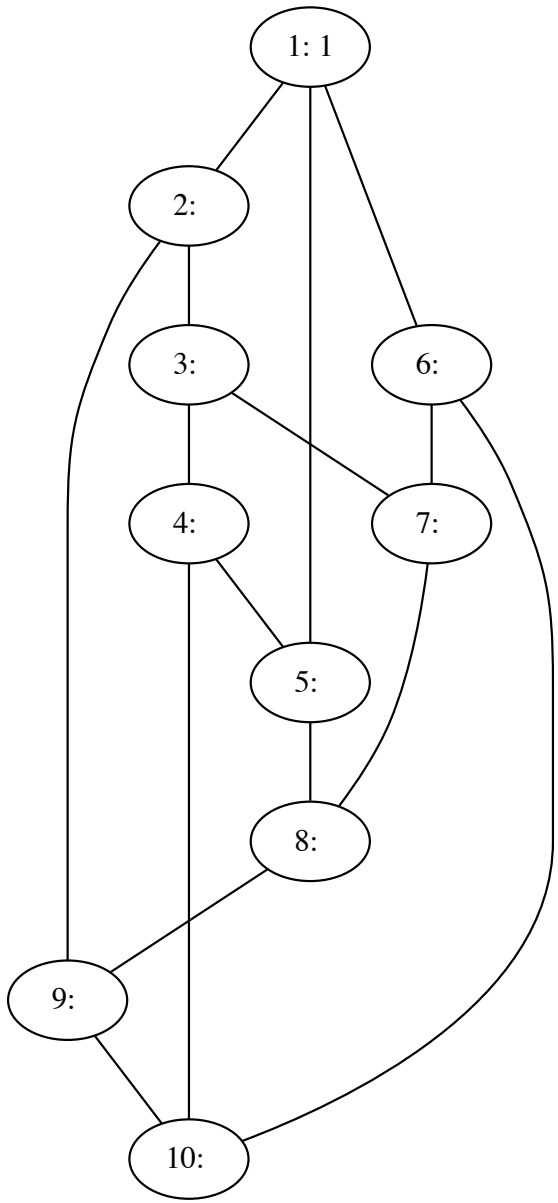
- Graphes orientés :
  - Detection de circuits (cycles orientés)
  - Tri topologique
  - Composantes fortement connexes

Plan

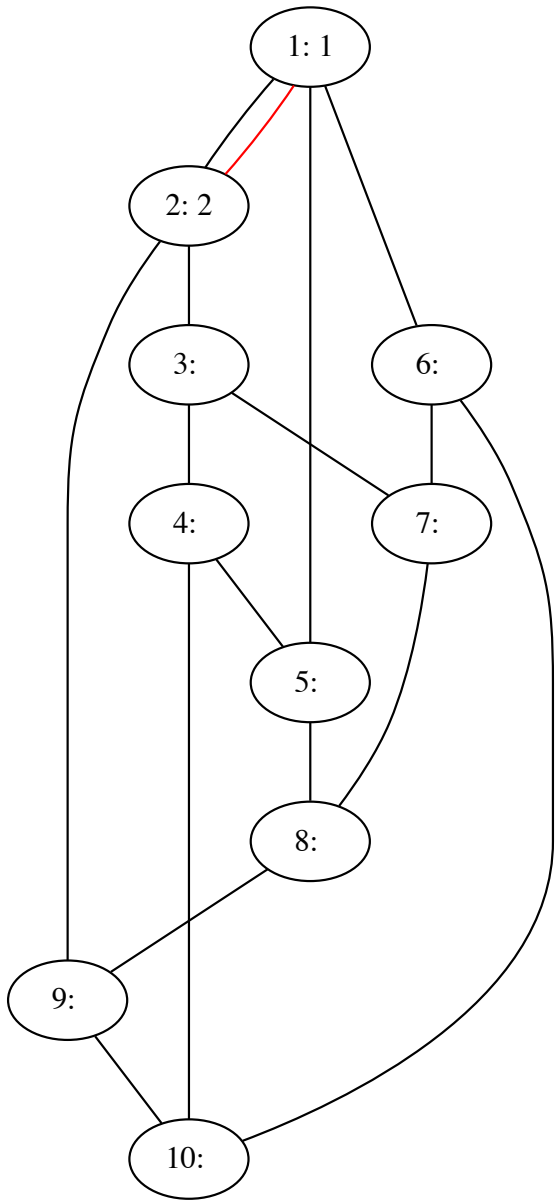
Parcours en profondeur récursif

```
times, parent = {}, {}


def dfsRecursive(adjLists):
    vertices = list(adjLists.keys())

    for x in vertices:
        times[x] = {'discovery': None, 'finish': None}

    i = 1
    for x in vertices:
        if times[x]['discovery'] is None:
            i = dfsRec(adjLists, x, i)
            i += 1
    return parent


def dfsRec(adjLists, root, i):
    global times, parent
    times[root]['discovery'] = i
    for x in adjLists[root]:
        if times[x]['discovery'] is None:
            parent[x] = root
            i = dfsRec(adjLists, x, i+1)
    i += 1
    times[root]['finish'] = i
    return i
```

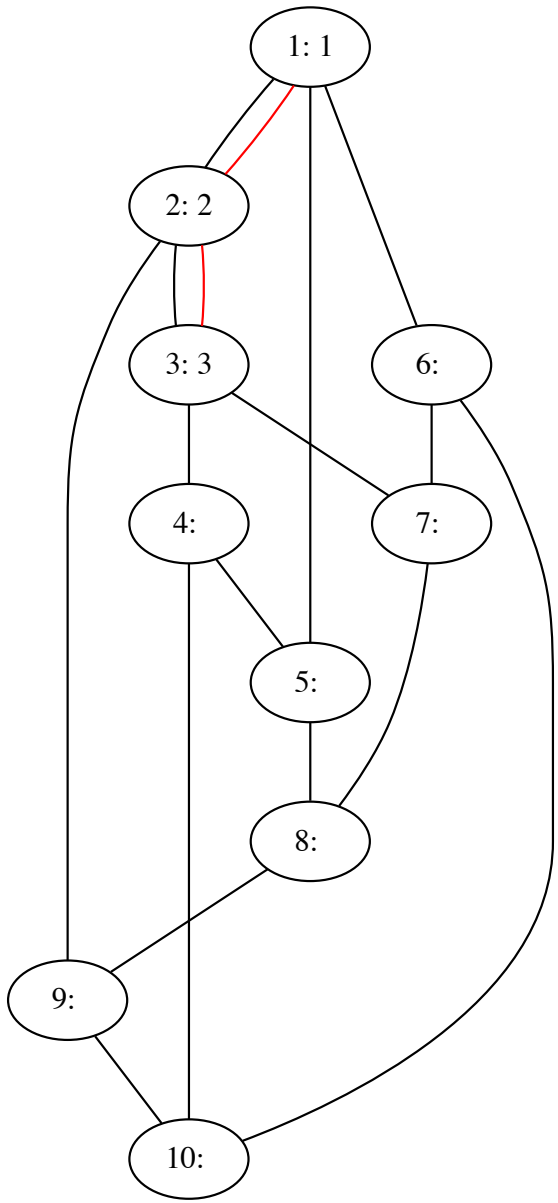
Parcours en profondeur récursif, exemple



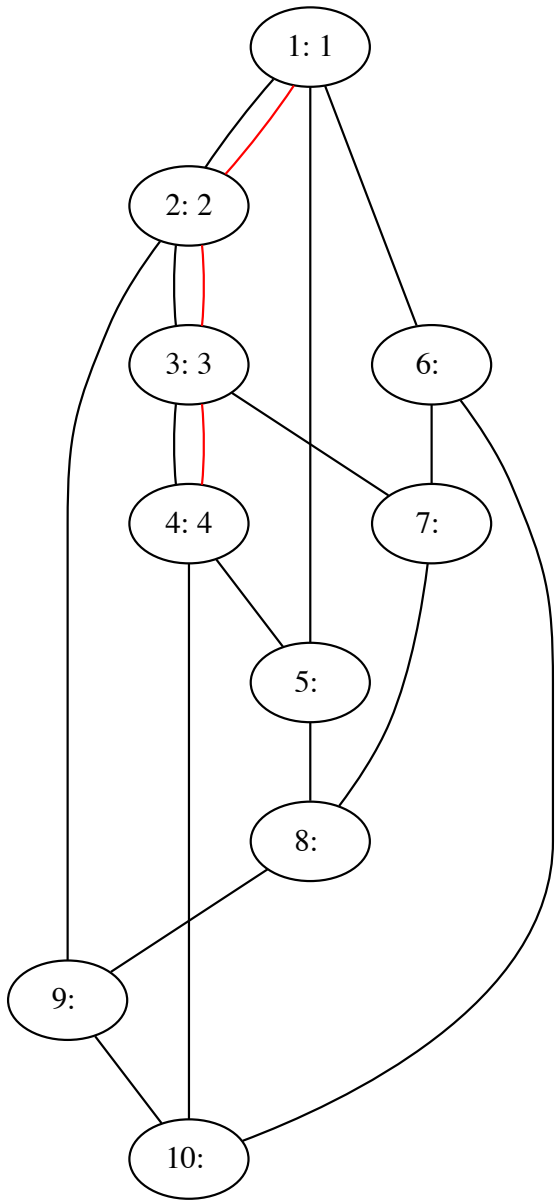
Parcours en profondeur récursif, exemple



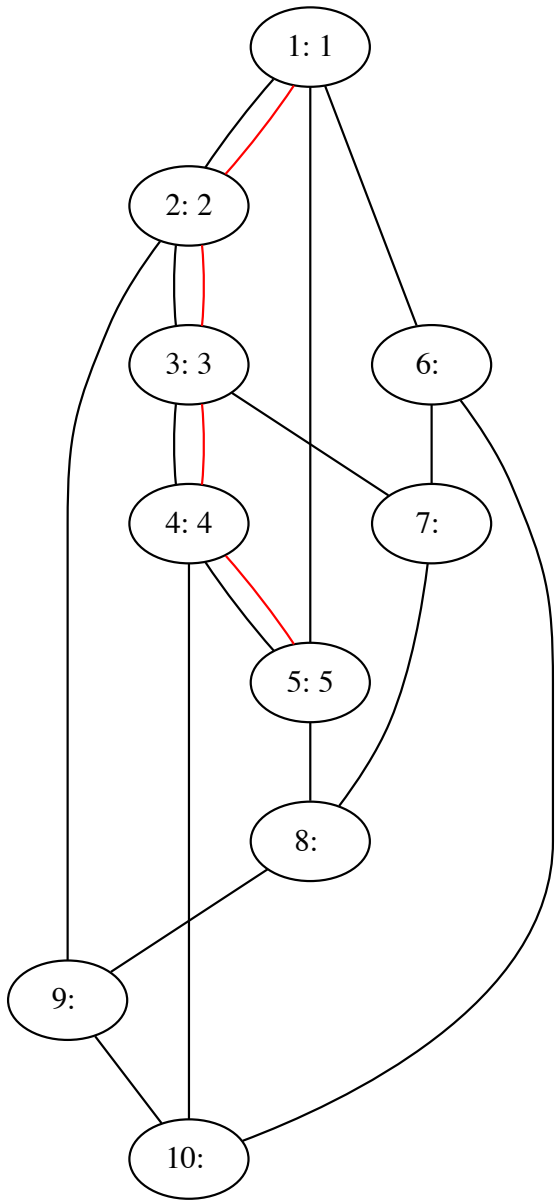
Parcours en profondeur récursif, exemple



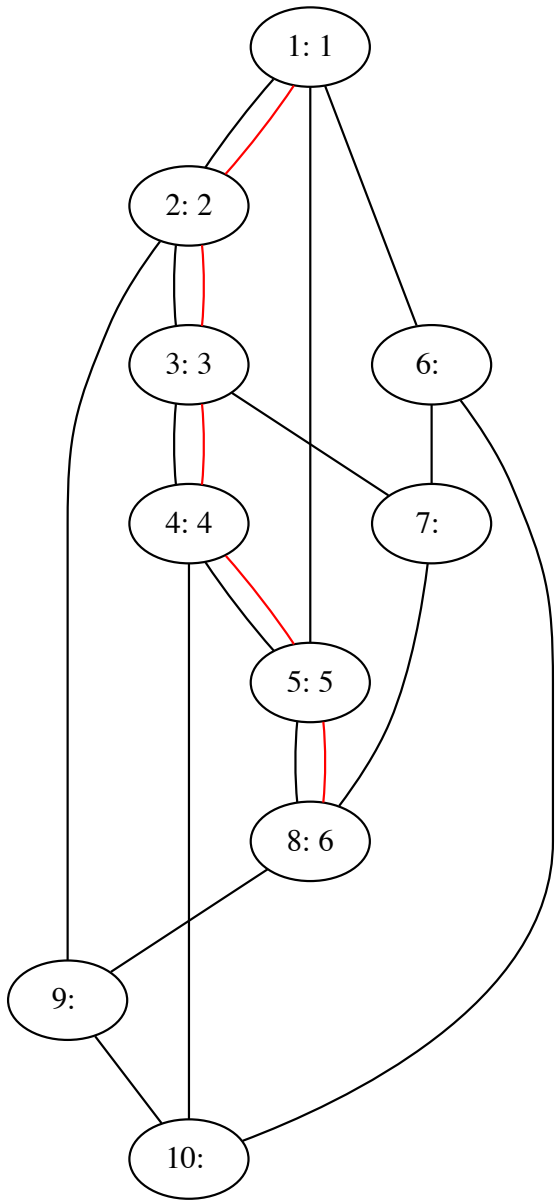
Parcours en profondeur récursif, exemple



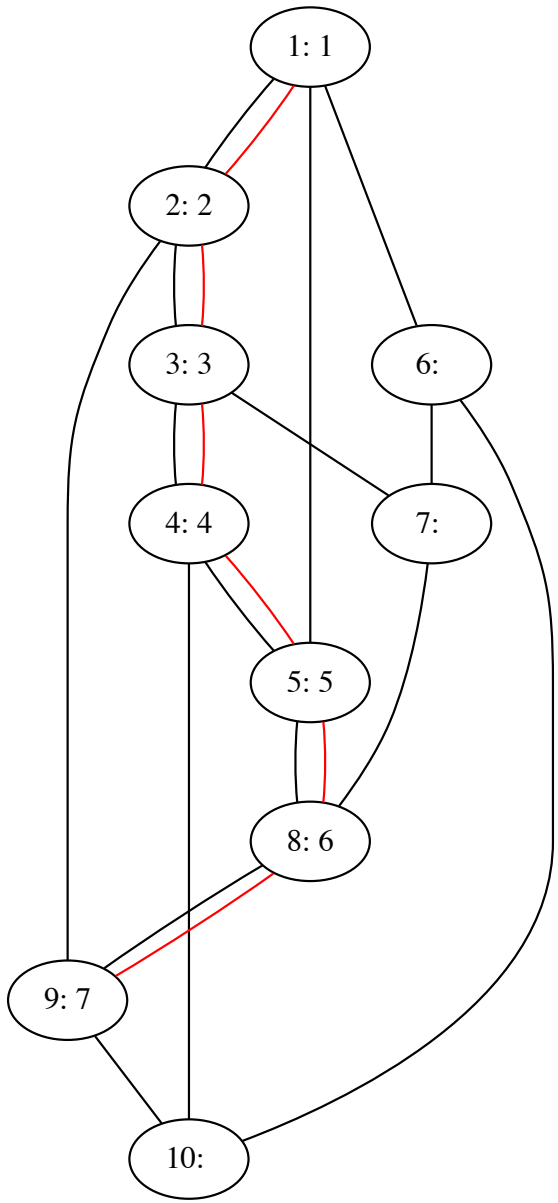
Parcours en profondeur récursif, exemple



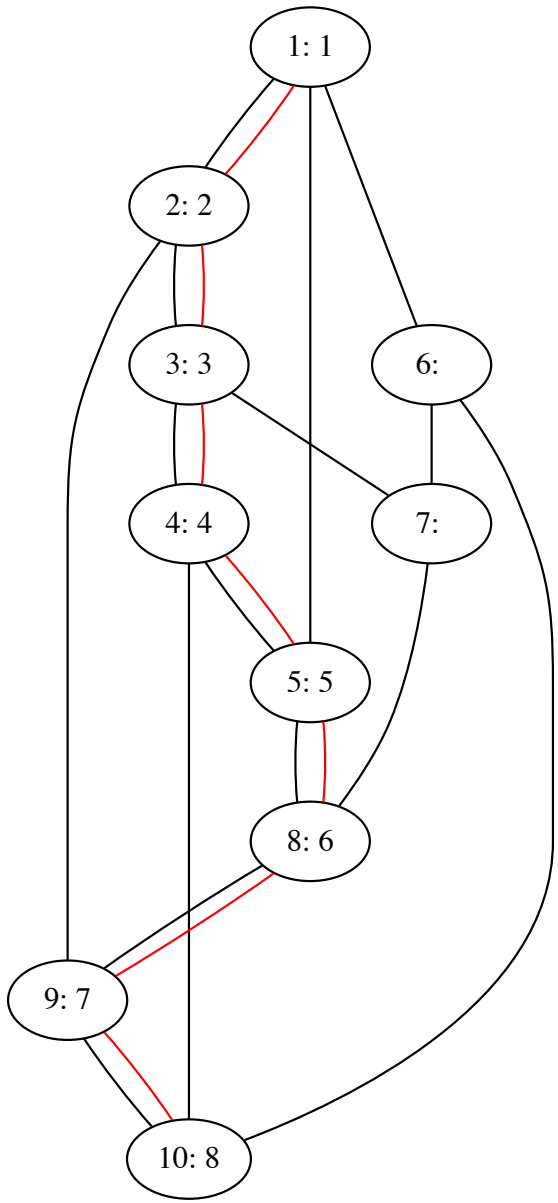
Parcours en profondeur récursif, exemple



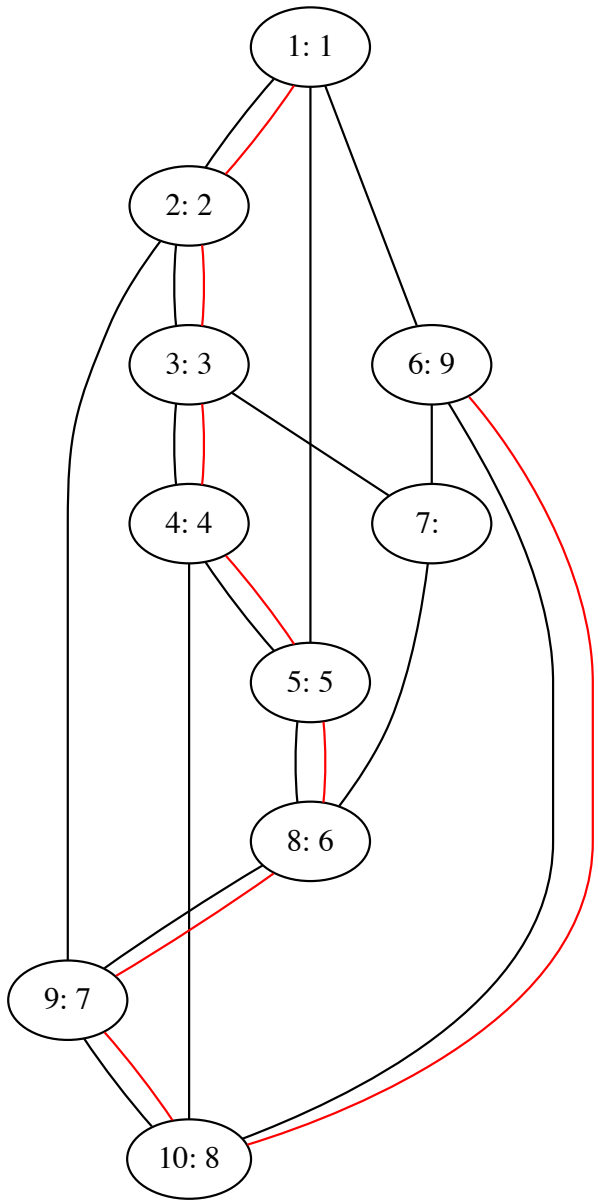
Parcours en profondeur récursif, exemple



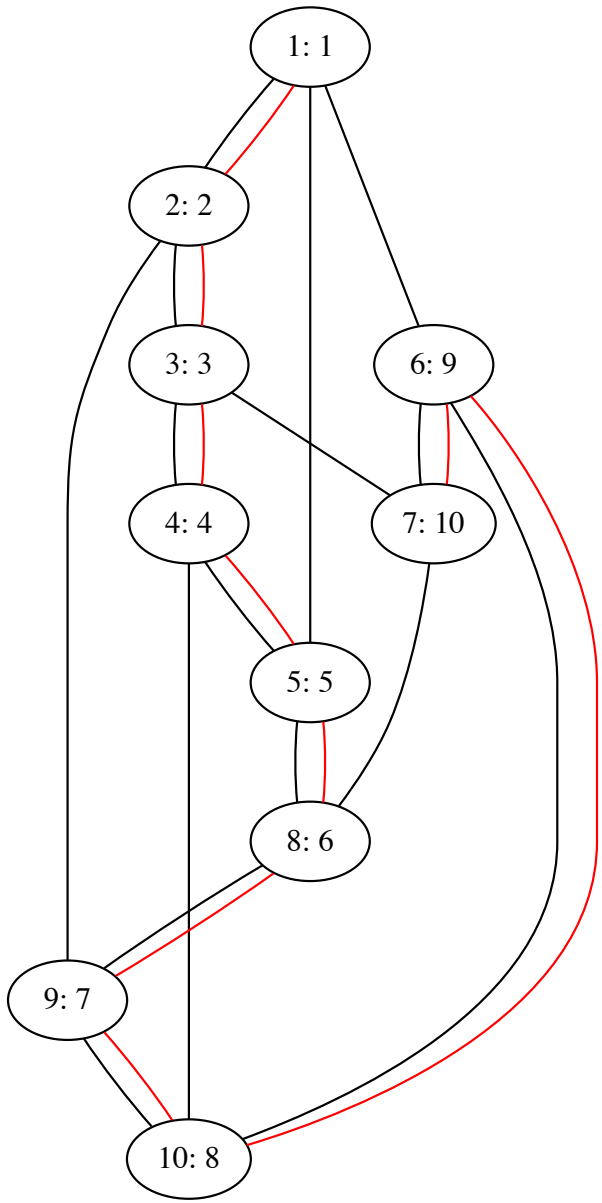
Parcours en profondeur récursif, exemple



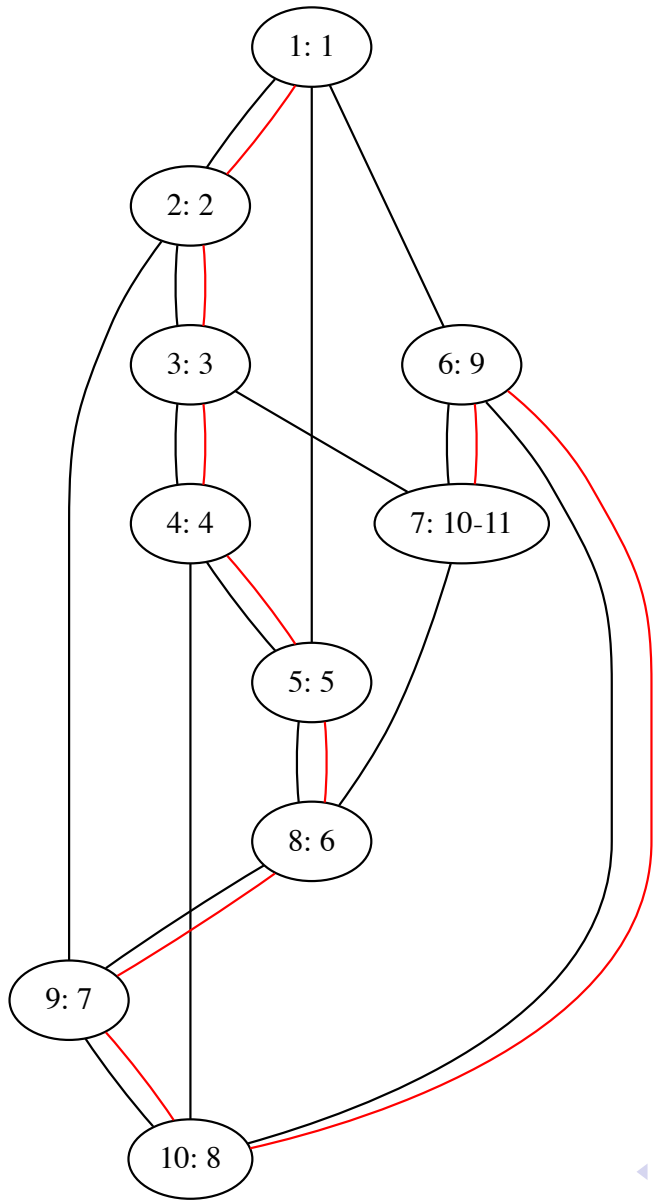
Parcours en profondeur récursif, exemple



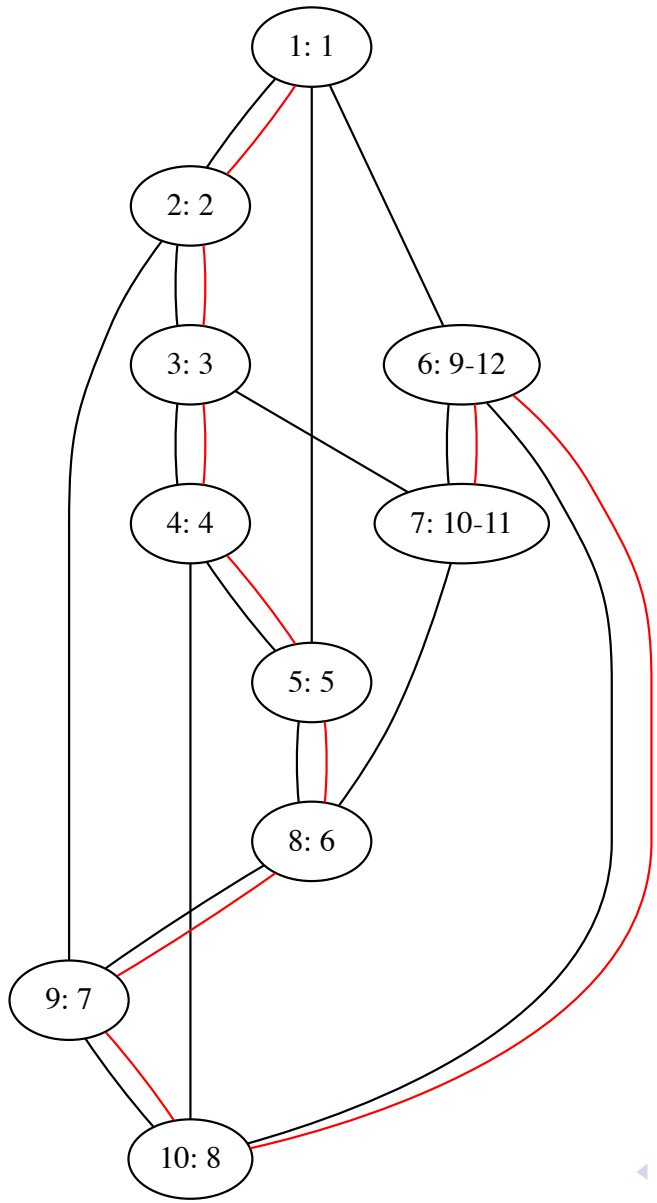
Parcours en profondeur récursif, exemple



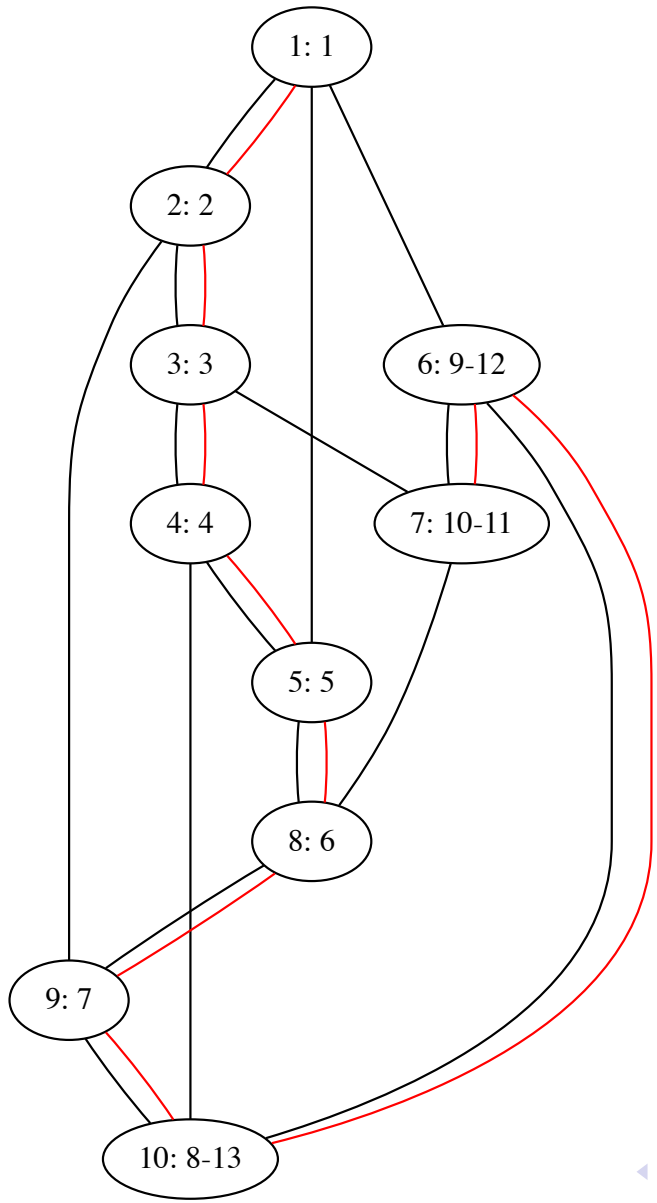
Parcours en profondeur récursif, exemple



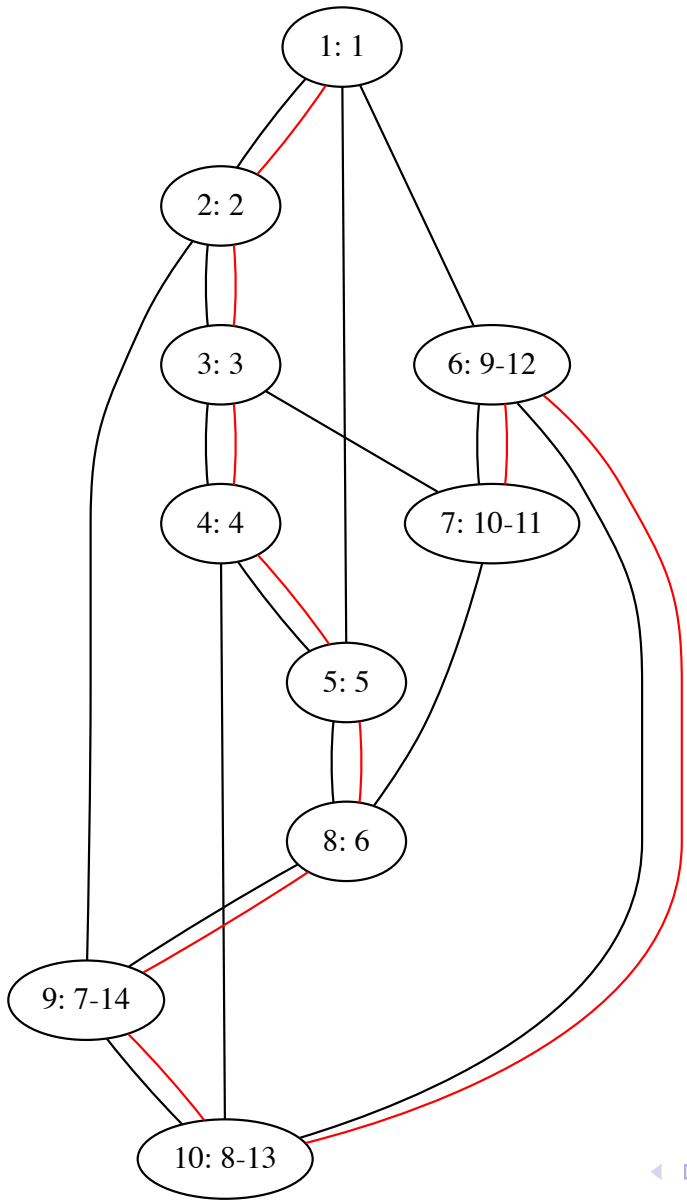
Parcours en profondeur récursif, exemple



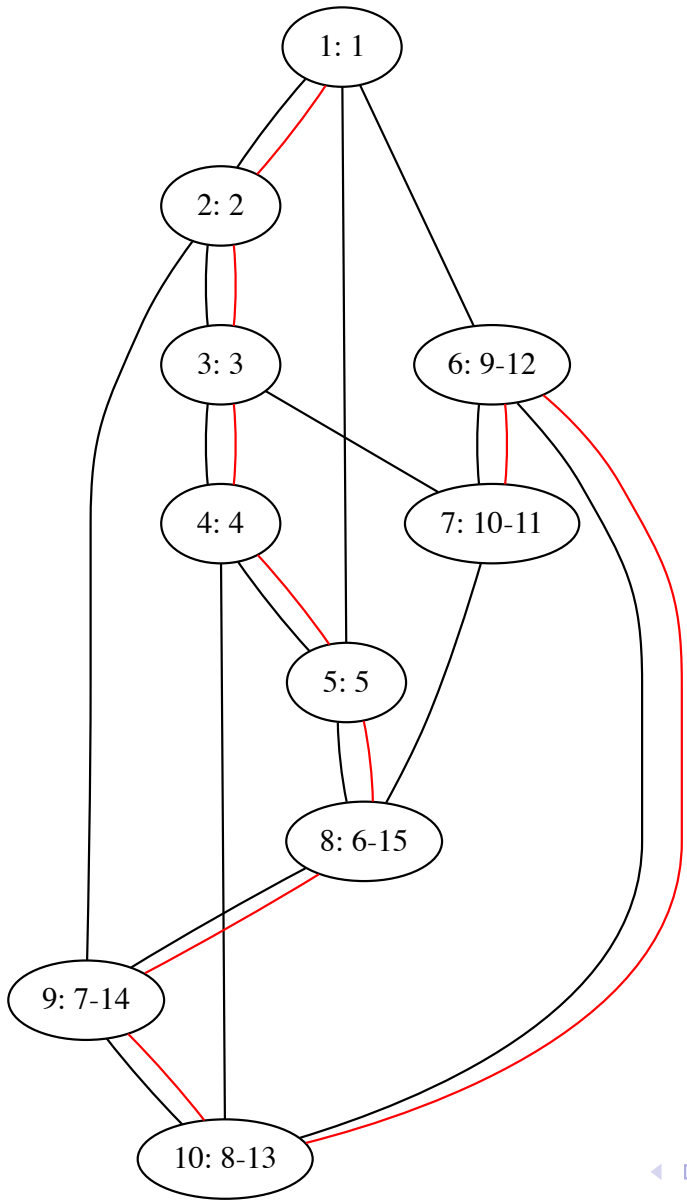
Parcours en profondeur récursif, exemple



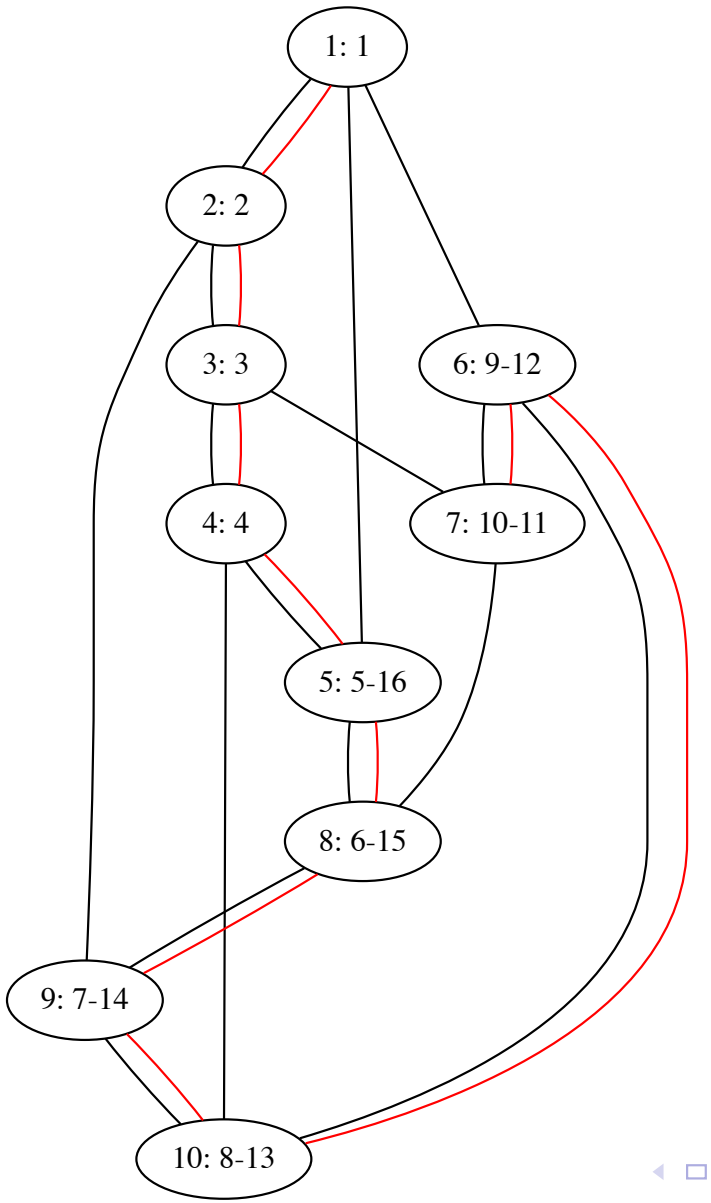
Parcours en profondeur récursif, exemple



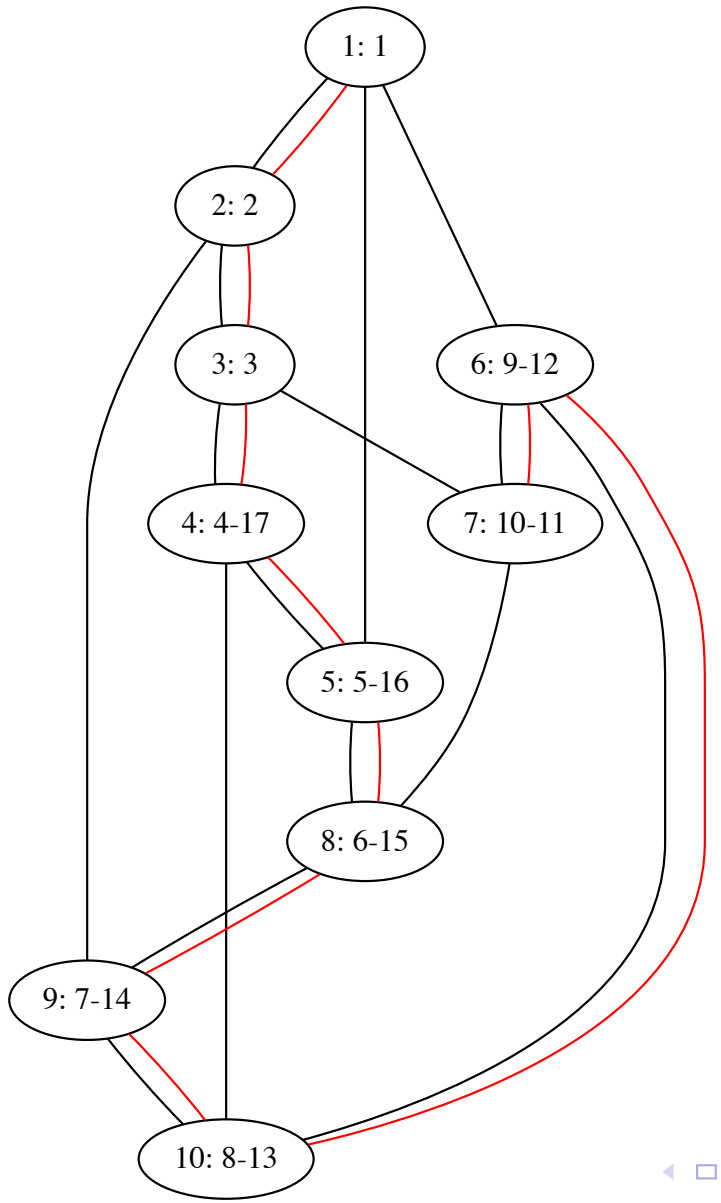
Parcours en profondeur récursif, exemple



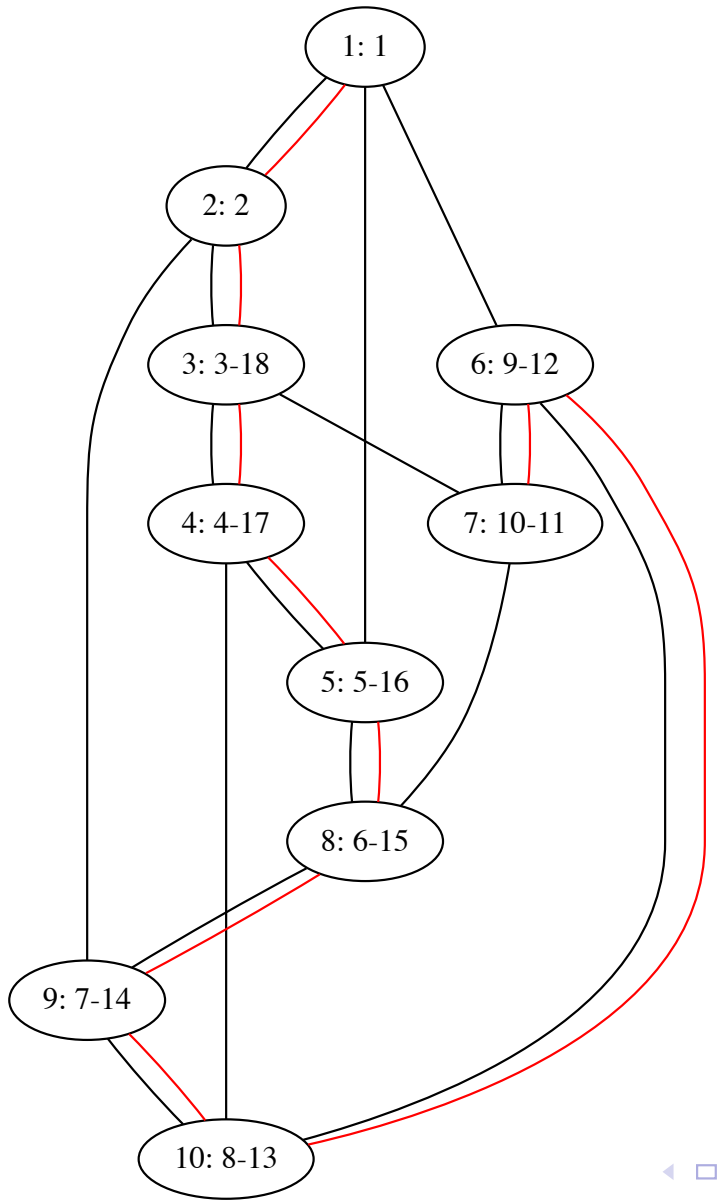
Parcours en profondeur récursif, exemple



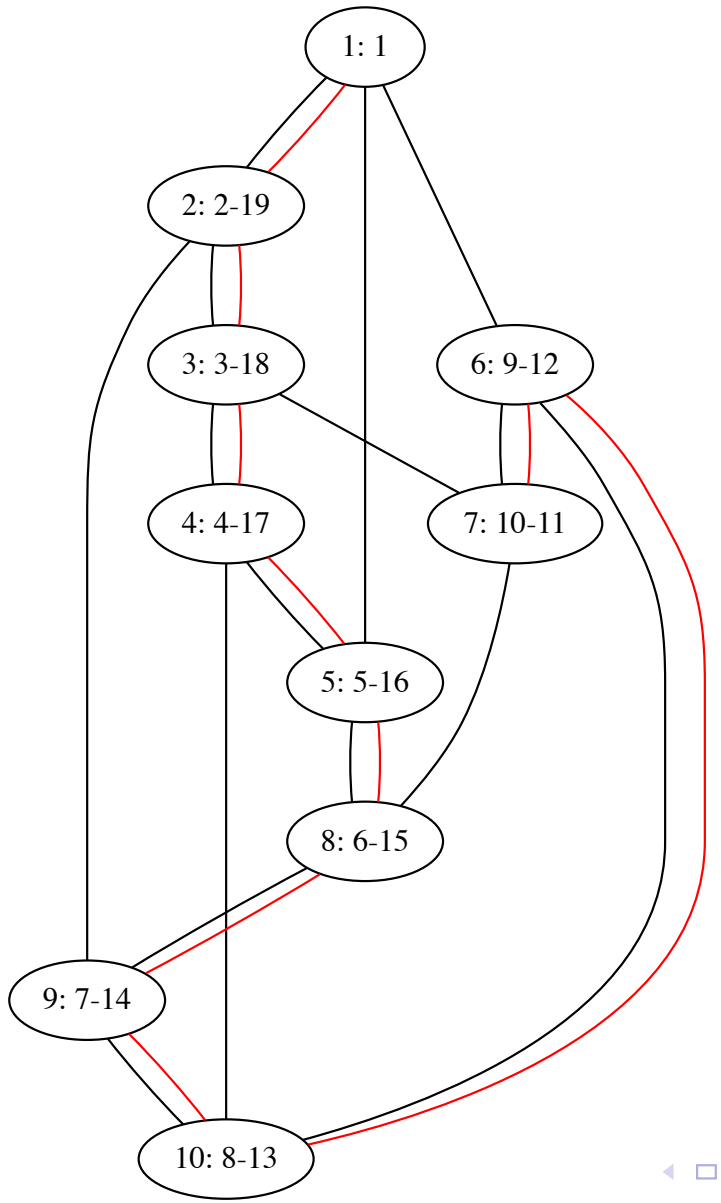
Parcours en profondeur récursif, exemple



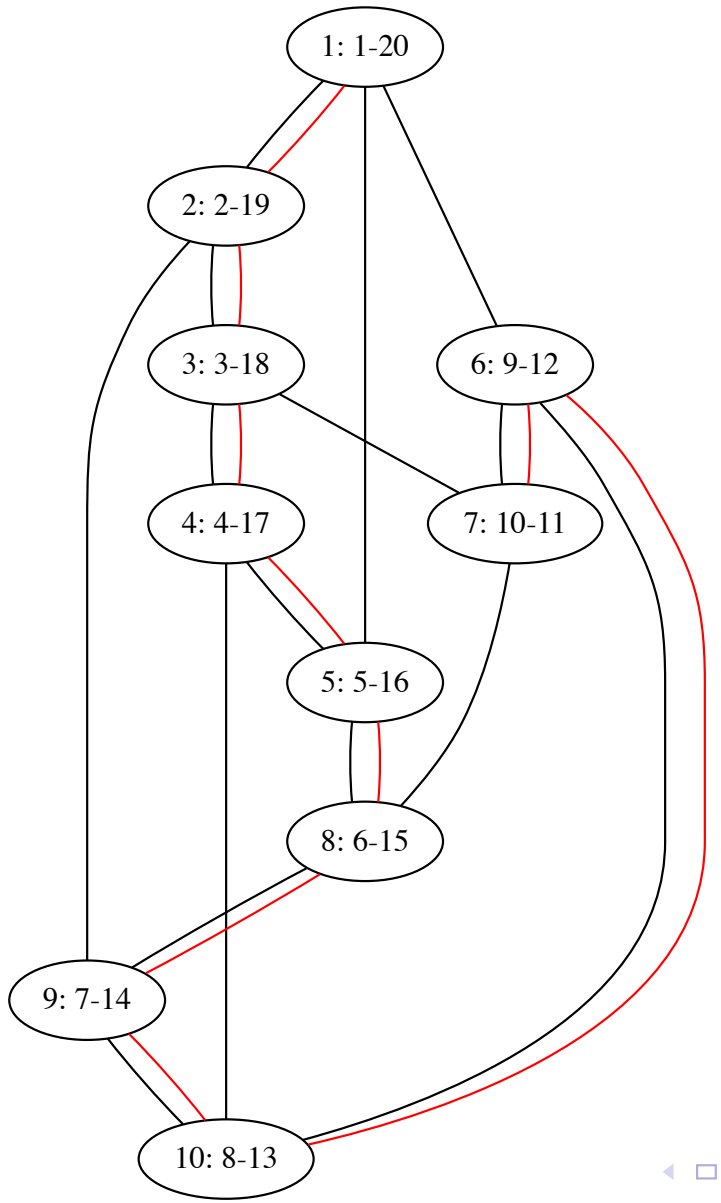
Parcours en profondeur récursif, exemple



Parcours en profondeur récursif, exemple



Parcours en profondeur récursif, exemple



Plan

Parcours en profondeur des graphes orientés

- Un graphe orienté est un couple (V, A) où
  - V est un ensemble fini de sommets,
  - $A \subseteq V \times V$. On appelle $(x, y) \in A$ un arc de (V, A).
- Parcours : comme pour le graphes non orientés, mais les listes/matrice d'adjacence peuvent ne pas être symétriques.

Classification des arcs

Après un parcours en profondeur de (V, A) retournant (V, T), un arc $(x, y) \in A$ peut être :

- un *arc descendant*, si

$$\texttt{times}[x]\texttt{['discovery']} < \texttt{times}[y]\texttt{['discovery']}$$
$$\texttt{times}[y]\texttt{['finish']} < \texttt{times}[x]\texttt{['finish']}$$

  On aura
  - un *arc de foret*, si $(x, y) \in T$
  - un *arc de transitivité*, sinon
- un *arc de retour*, si

$$\texttt{times}[x]\texttt{['discovery']} > \texttt{times}[y]\texttt{['discovery']}$$
$$\texttt{times}[x]\texttt{['finish']} < \texttt{times}[y]\texttt{['finish']}$$

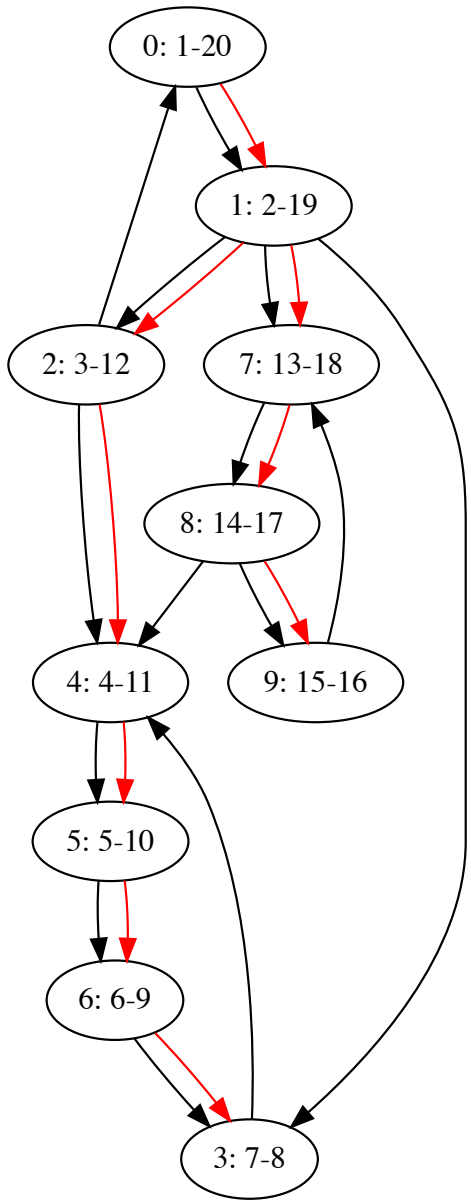
- un *arc traversier*, autrement, c'est-à-dire, si

$$\texttt{times}[x]\texttt{['discovery']} > \texttt{times}[y]\texttt{['discovery']}$$
$$\texttt{times}[x]\texttt{['finish']} > \texttt{times}[y]\texttt{['finish']}$$

  On aura, dans ce cas :

$$\texttt{times}[x]\texttt{['discovery']} > \texttt{times}[y]\texttt{['finish']}$$

Exemple



Remarques

- Pas d'arc traversiers si A est symétrique
  (c'est-à-dire $(x, y) \in A$ implique $(y, x) \in A$)
- Un graphe orienté est acyclique ssi, après un parcours en profondeurs, on a pas de arcs de retour.
  - Détection de cycles devient détection des arcs de retour.

Tri topologique et DFS

Definition

Un *tri topologique* de (V, A) est une fonction injective

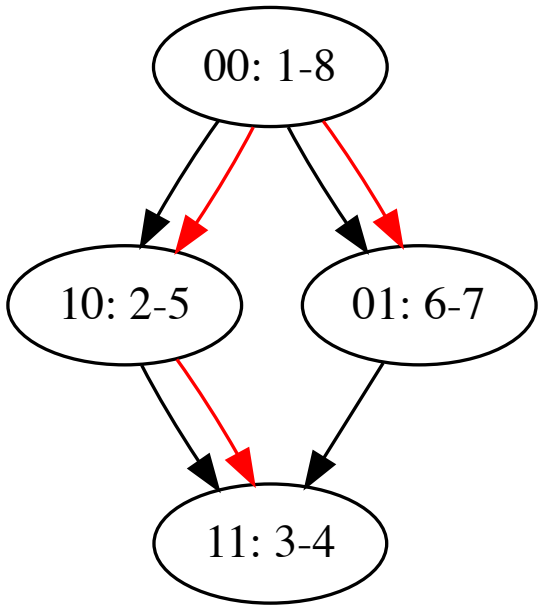
$$\pi : V \to \mathbb{Z}$$

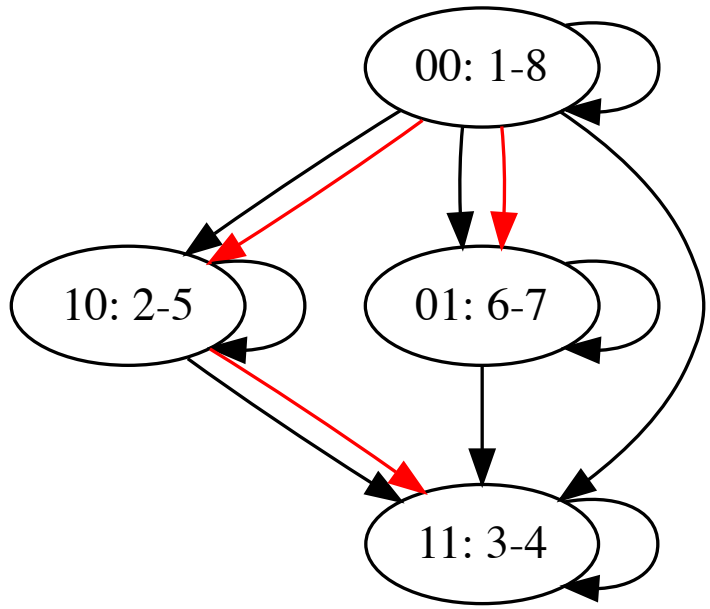
telle que $(x, y) \in A$ implique $\pi(x) < \pi(y)$.

Un tri topologique existe si et seulement si (V, A) est sans circuits.

Si (V, A) est sans circuits, alors l'opposé du temps de fin donne un tri topologique de (V, A).

Exemples, algèbre de Boole avec 4 éléments





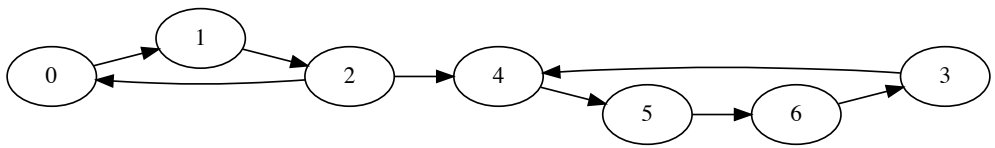
$$v(00) = -8, \quad v(01) : -7, \quad v(10) = -5, \quad v(11) = -4\,.$$

Composantes fortement connexes

Definition

Soit (V, A) un graphe orienté. $x, y \in V$ appartiennent à la même *composante fortement connexe* si il existe un chemin de l'un à l'autre, et vice-versa.

Exemple :



- Deux composantes fortement connexes :

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

- Une seule composante connexe.

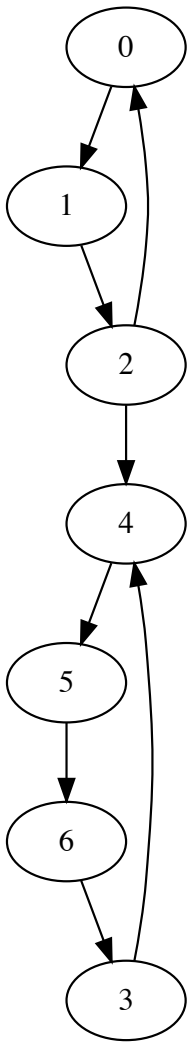
DFS et calcul des composantes fortement connexes

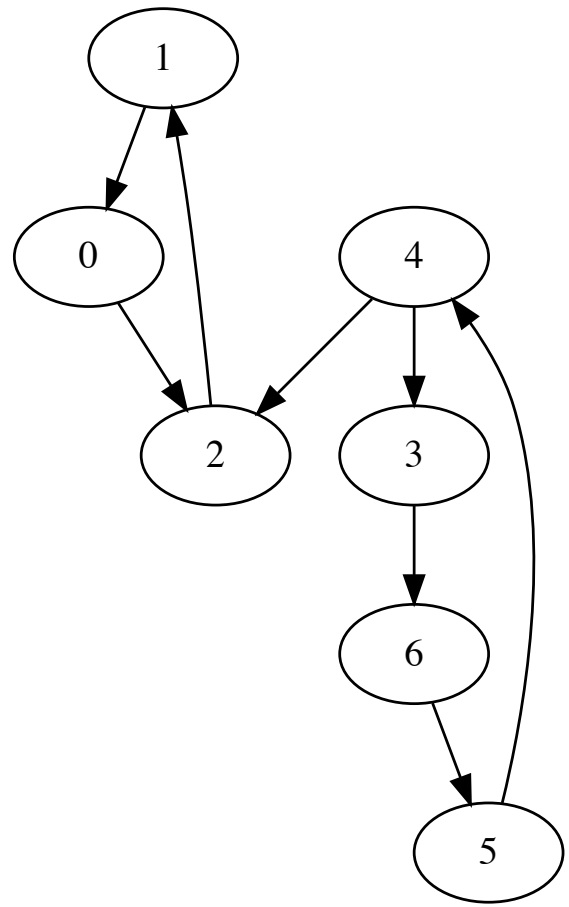
- On fait un parcours du graphe orienté (V, A^{rev}),
  où $(x, y) \in A^{rev}$ ssi $(y, x) \in A$.
  On ainsi calcule

$$f(v) = \texttt{times[}v\texttt{]['finish']},$$

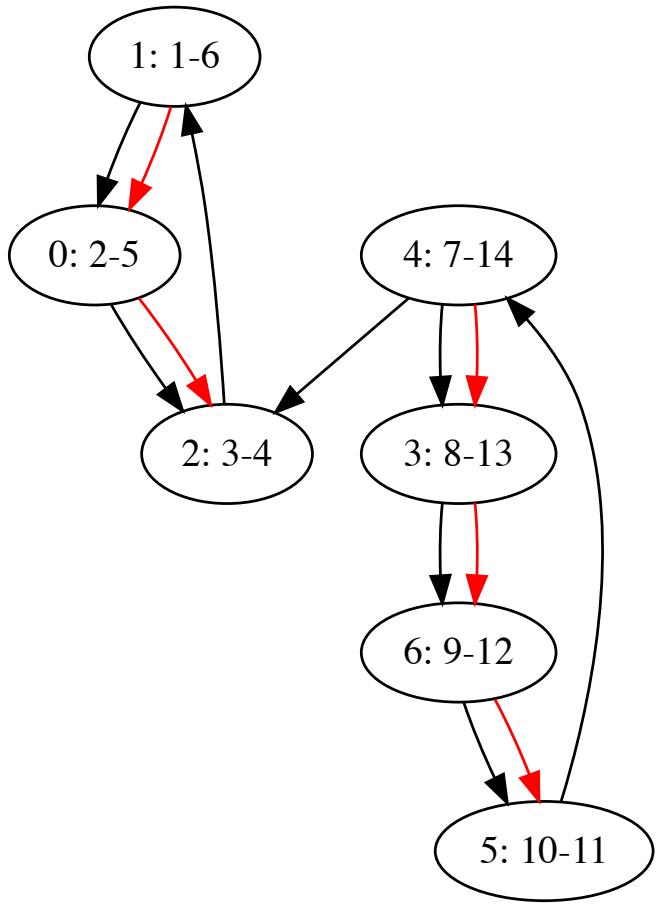
  pour tout $v \in V$.
- On fait un deuxième parcours, cette fois ci de (V, A).
  Le sommets de V sont ordonnées par f décroissant.
- Deux sommets sont dans la même composante connexe ssi il sont relié à la même racine.

Exemple





Exemple



Ordre : $4, 3, 6, 5, 1, 0, 2$.

Exemple