Parcours et structures linéaires

Christophe Viroulaud

Terminale - NSI

Algo 21

Parcours et structures linéaires

Rappels

Les structures linéaires

Représentation d'un graphe en mémoire

Parcours en profondeur

Parcours en largeur

Principe

Implémentation

L'algorithme de parcours en profondeur est récursif. Il est alors naturel de pouvoir l'écrire à nouveau avec une structure linéaire : la **pile**.

Comment implémenter les algorithmes de parcours avec des structures linéaires ?

Sommaire

1. Rappels
1.1 Les structures linéaires
1.2 Représentation d'un graphe en mémoire

2. Parcours en profondeur

3. Parcours en largeur

Rappels : Les structures linéaires

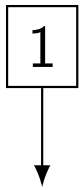


FIGURE 1 – Un nœud

Activité 1 :

1. Créer un fichier **`structures.py`**
2. Écrire la classe **`Noeud`** et son constructeur qui, lors de l'instanciation, initialisera deux attributs :
   - **`nom: int`** le nom du nœud,
   - **`suivant: Noeud`** le successeur dans la structure.
   - Écrire les accesseurs et les mutateurs des attributs.

Parcours et structures linéaires

Rappels

Les structures linéaires

Représentation d'un graphe en mémoire

Parcours en profondeur

Parcours en largeur

Principe

Implémentation

Correction

```
1  class Noeud:
2      def __init__(self, nom: int, suivant: object):
3          self.nom = nom
4          self.suivant = suivant
5
6      def get_nom(self) -> int:
7          return self.nom
8
9      def set_nom(self, n: int) -> None:
10         self.nom = n
11
12     def get_suivant(self) -> object:
13         return self.suivant
14
15     def set_suivant(self, s: object) -> None:
16         self.suivant = s
```

Parcours et structures linéaires

Rappels

Les structures linéaires

Représentation d'un graphe en mémoire

Parcours en profondeur

Parcours en largeur

Principe

Implémentation

Rappels : les structures linéaires

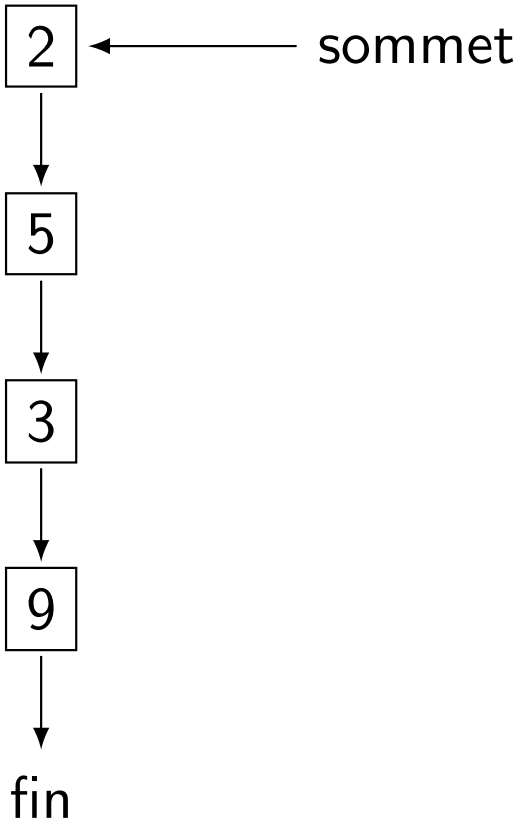


FIGURE 2 – Pile

Activité 2 :

1. Écrire la classe **`Pile`** et son constructeur qui initialisera l'attribut **`sommet`** à **`None`**.
2. Écrire la méthode **`est_vide`** qui renverra **`True`** si la pile est vide.
3. Écrire la méthode **`empiler`** qui empilera le nœud nommé **`n: int`** passé en paramètre.
4. Écrire la méthode **`depiler`** qui dépilera le nœud au sommet de la pile et renverra son nom ou -1 si la pile est vide.

Correction

```
1  class Pile:
2      def __init__(self):
3          self.sommet = None
4
5      def est_vide(self) -> bool:
6          return self.sommet is None
```

```
1      def empiler(self, n: int) -> None:
2          self.sommet = Noeud(n, self.sommet)
3
4      def depiler(self) -> int:
5          res = -1
6          if not self.est_vide():
7              res = self.sommet.get_nom()
8              self.sommet = self.sommet.get_suivant()
9          return res
```

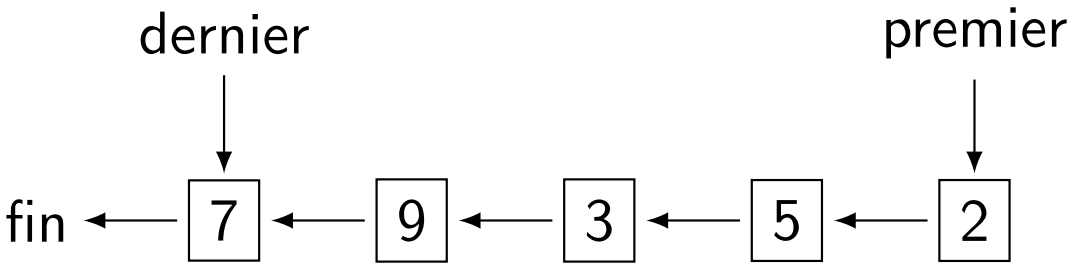


FIGURE 3 – File

Activité 3 :

1. Écrire la classe **`File`** et son constructeur qui initialisera à **`None`** deux attributs :
   - **`premier`**,
   - **`dernier`**.
2. Écrire la méthode **`est_vide`** qui renverra **`True`** si la file est vide.
3. Écrire la méthode **`enfiler`** qui enfilera le nœud nommé **`n: int`** passé en paramètre.
4. Écrire la méthode **`defiler`** qui défilera le nœud en première place de la file et renverra son nom ou -1 si la file est vide.

Correction

```
1  class File:
2      def __init__(self):
3          self.premier = None
4          self.dernier = None
5
6      def est_vide(self) -> bool:
7          return self.premier is None
```

```
1     def enfiler(self, n: int) -> None:
2         nouveau = Noeud(n, None)
3         if self.est_vide():
4             self.premier = nouveau
5         else:
6             self.dernier.set_suivant(nouveau)
7         self.dernier = nouveau
8
9     def defiler(self) -> int:
10        res = -1
11        if not self.est_vide():
12            res = self.premier.get_nom()
13            self.premier = self.premier.get_suivant()
14        return res
```

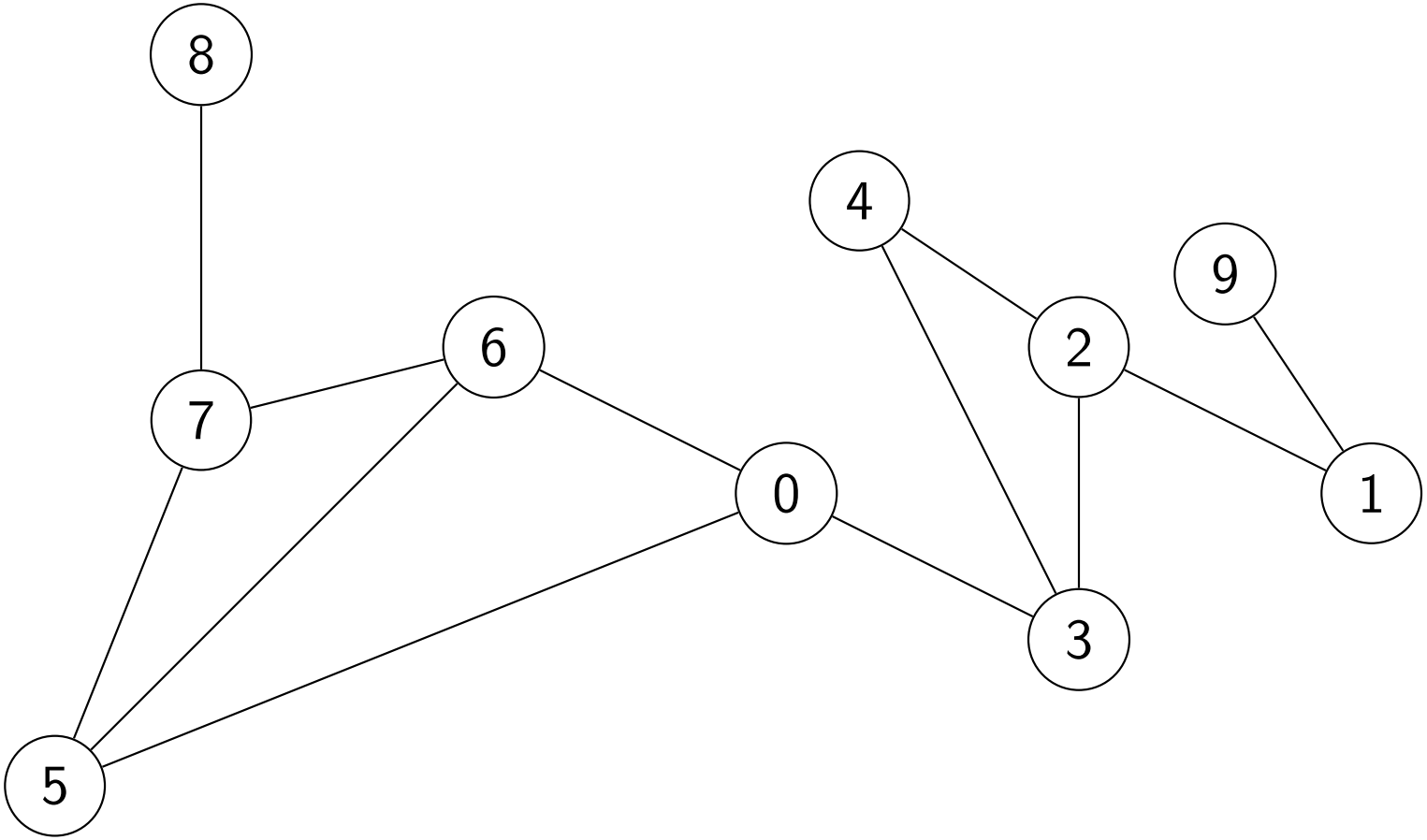
Sommaire

1. Rappels
1.1 Les structures linéaires
1.2 Représentation d'un graphe en mémoire

2. Parcours en profondeur

3. Parcours en largeur

Représentation d'un graphe en mémoire



Activité 4 :

1. Créer le fichier `parcours.py`
2. Écrire la liste d'adjacence du graphe.

Correction

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

Sommaire

1. Rappels

2. Parcours en profondeur

3. Parcours en largeur

Parcours en profondeur

Parcours et structures linéaires

Rappels

Les structures linéaires

Représentation d'un graphe en mémoire

Parcours en profondeur

Parcours en largeur

Principe

Implémentation

À retenir

Lors d'un parcours en profondeur on repart du nœud en cours de visite pour continuer le parcours. On utilise alors une **pile** pour l'implémenter.

Activité 5 :

1. Écrire la fonction `dfs(graphe: list)` $\rightarrow$ `list` qui effectue un parcours en profondeur en utilisant une pile. La fonction renverra la liste ordonnée des sommets parcourus.
2. Écrire la fonction `dfs_aretes(graphe: list)` $\rightarrow$ `list` qui renvoie les liste ordonnée des arêtes parcourues. La fonction colorera les sommets en :
   - `BLANC` au départ,
   - `GRIS` lors de l'empilement,
   - `NOIR` quand son parcours est terminé.

Correction

```python
def dfs(graphe: list) -> list:
    parcours = []
    p = Pile()
    p.empiler(0)
    while not p.est_vide():
        en_cours = p.depiler()
        if en_cours not in parcours:
            parcours.append(en_cours)

            for voisin in graphe[en_cours]:
                p.empiler(voisin)
    return parcours
```

```
>>> dfs(graphe)
[0, 6, 7, 8, 5, 3, 4, 2, 1, 9]
```

Code 1 – Appel de la fonction

Parcours et structures linéaires

Rappels

Les structures linéaires

Représentation d'un graphe en mémoire

Parcours en profondeur

Parcours en largeur

Principe

Implémentation

Correction

```
1  def dfs_aretes(graphe: list) -> list:
2      parcours = []
3      etats = [BLANC for _ in range(len(graphe))]
4      p = Pile()
5      p.empiler(0)
6      while not p.est_vide():
7          en_cours = p.depiler()
8          for voisin in graphe[en_cours]:
9              if etats[voisin] == BLANC:
10                 etats[voisin] = GRIS
11                 p.empiler(voisin)
12             # stockage arête parcourue
13             parcours.append((en_cours, voisin))
14
15         etats[en_cours] = NOIR
16     return parcours
```

Parcours et structures linéaires

Rappels

Les structures linéaires

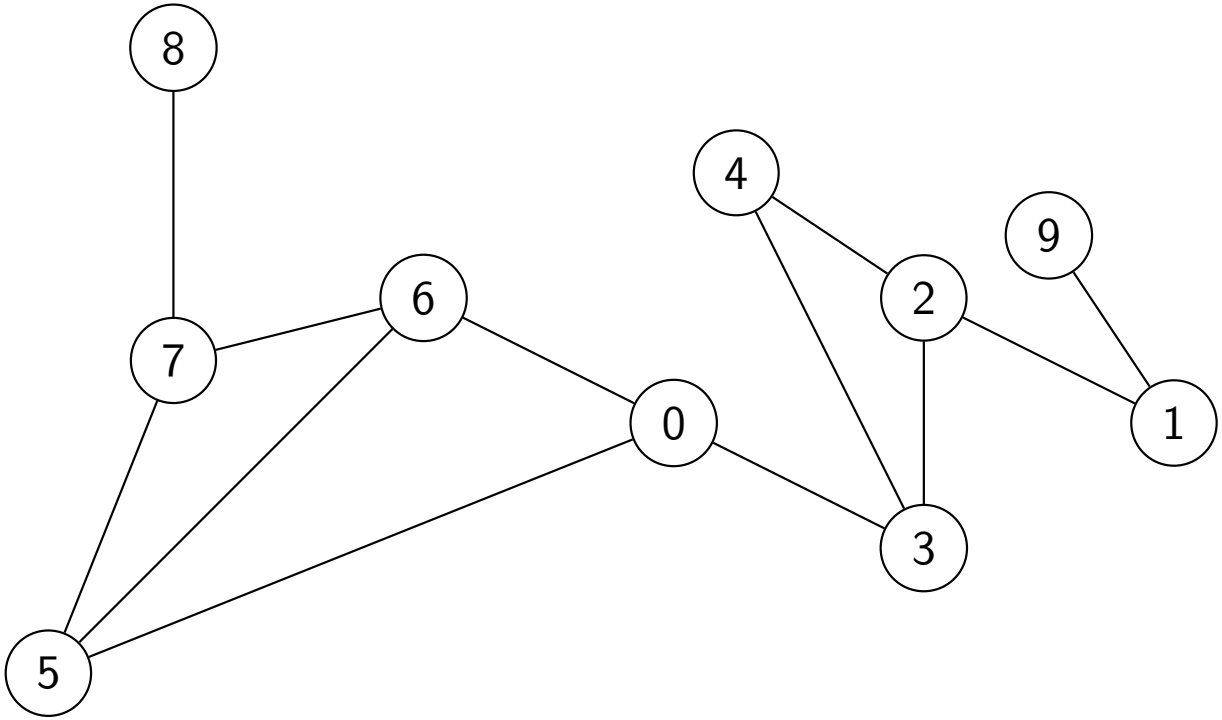
Représentation d'un graphe en mémoire

Parcours en profondeur

Parcours en largeur

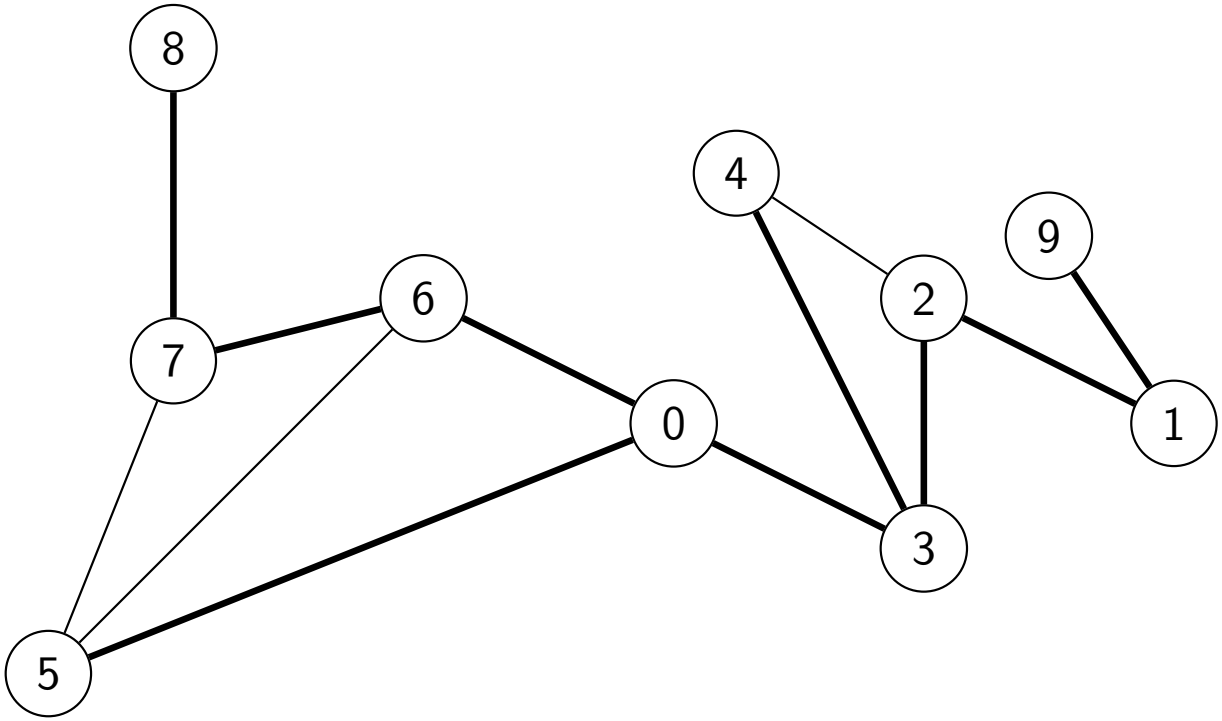
Principe

Implémentation




```
1 >>> dfs_aretes(graphe)
2 [(0, 3), (0, 5), (0, 6), (6, 0), (6, 5), (6, 7),
   (7, 5), (7, 6), (7, 8), (8, 7), (5, 0), (5, 6),
   (5, 7), (3, 0), (3, 2), (3, 4), (4, 2), (4, 3),
   (2, 1), (2, 3), (2, 4), (1, 2), (1, 9), (9, 1)]
```

Code 2 – Chaque sommet n'est empilé qu'une seule fois. Chaque arête est parcourue deux fois (graphe non orienté).




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

Observation

Si on stocke les arêtes seulement quand on empile les voisins, elles ne sont pas toutes visitées.

Sommaire

1. Rappels

2. Parcours en profondeur

3. Parcours en largeur
3.1 Principe
3.2 Implémentation

Parcours en largeur - Principe

Parcours et structures linéaires

Rappels

Les structures linéaires

Représentation d'un graphe en mémoire

Parcours en profondeur

Parcours en largeur

Principe

Implémentation

À retenir

Le parcours en largeur visite tous les sommets au rang `n` avant ceux au rang `n+1`. On utilise une **file** pour l'implémenter.

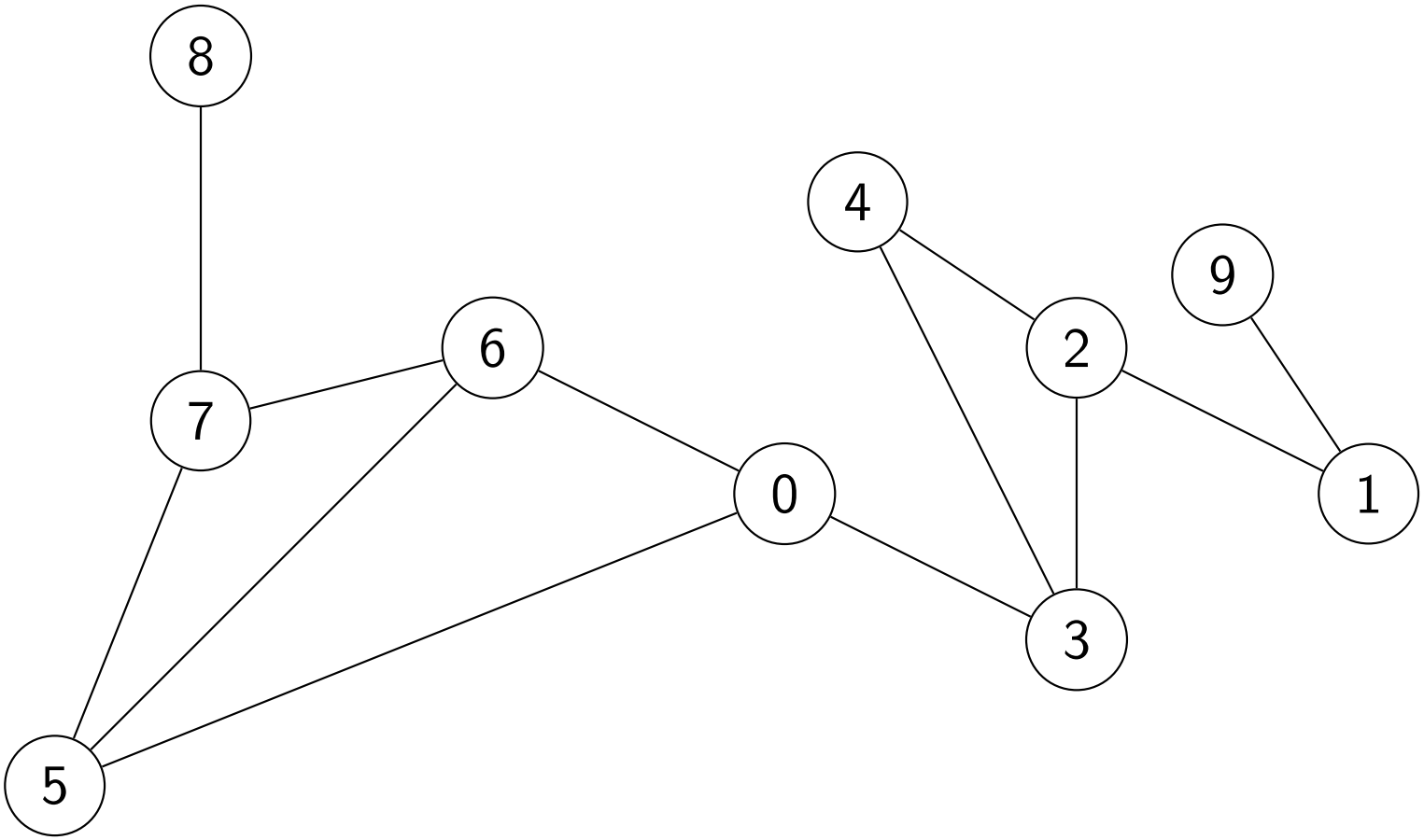


FIGURE 4 – Départ

File : 0

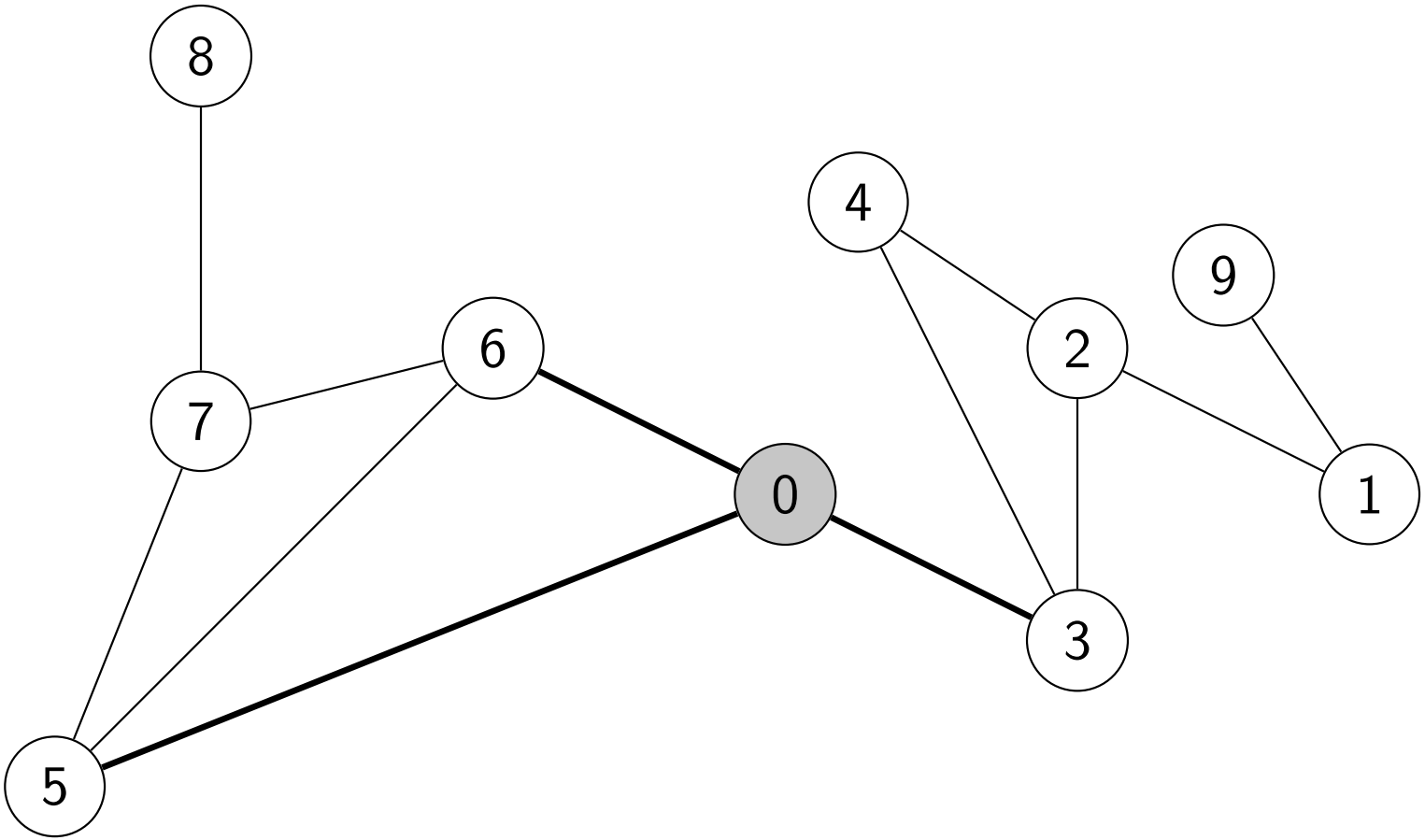


FIGURE 5 – On enfile les voisins à distance 1.

File : 6 → 5 → 3

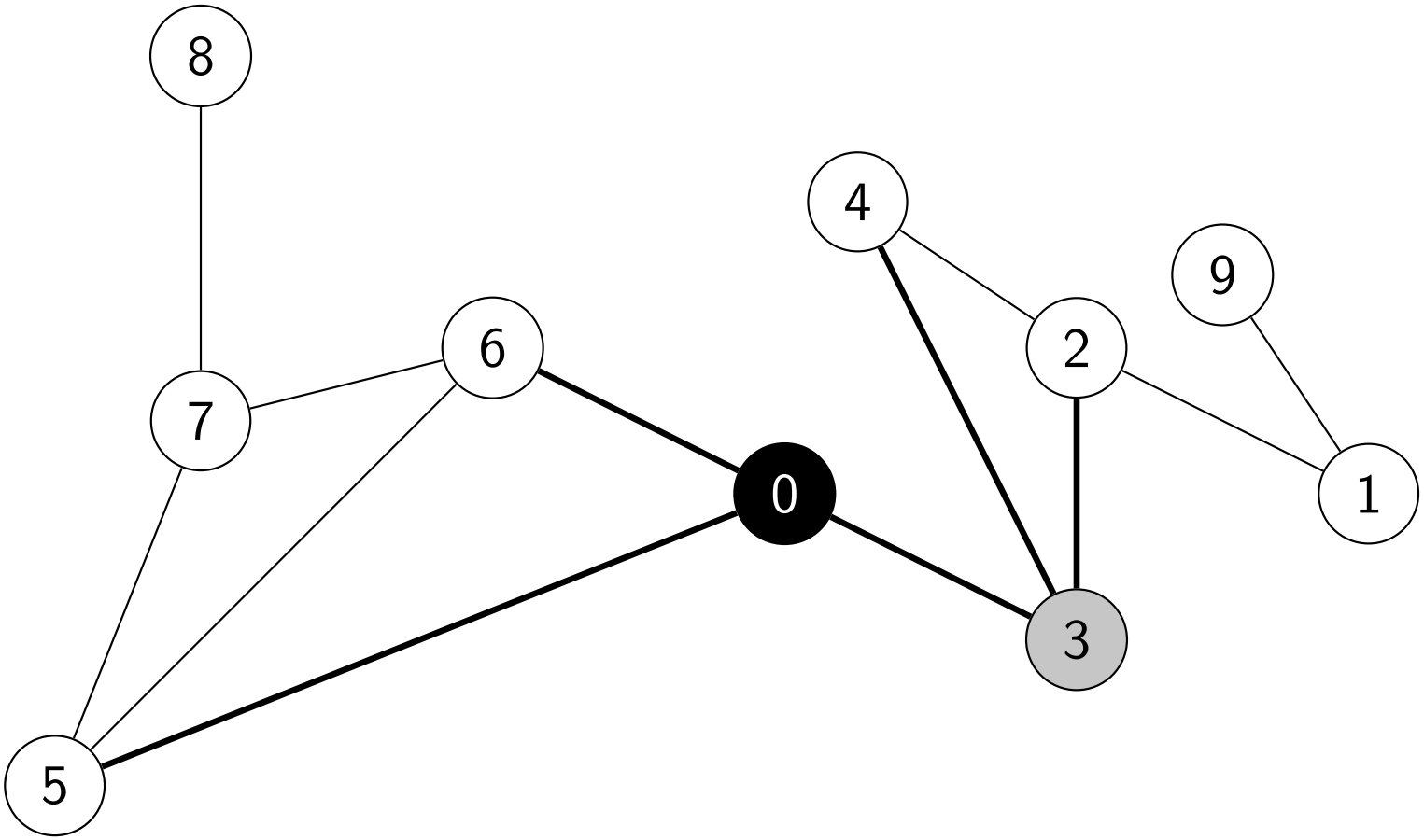


FIGURE 6 – On enfile les voisins à distance 2.

File : $4 \rightarrow 2 \rightarrow 6 \rightarrow 5$

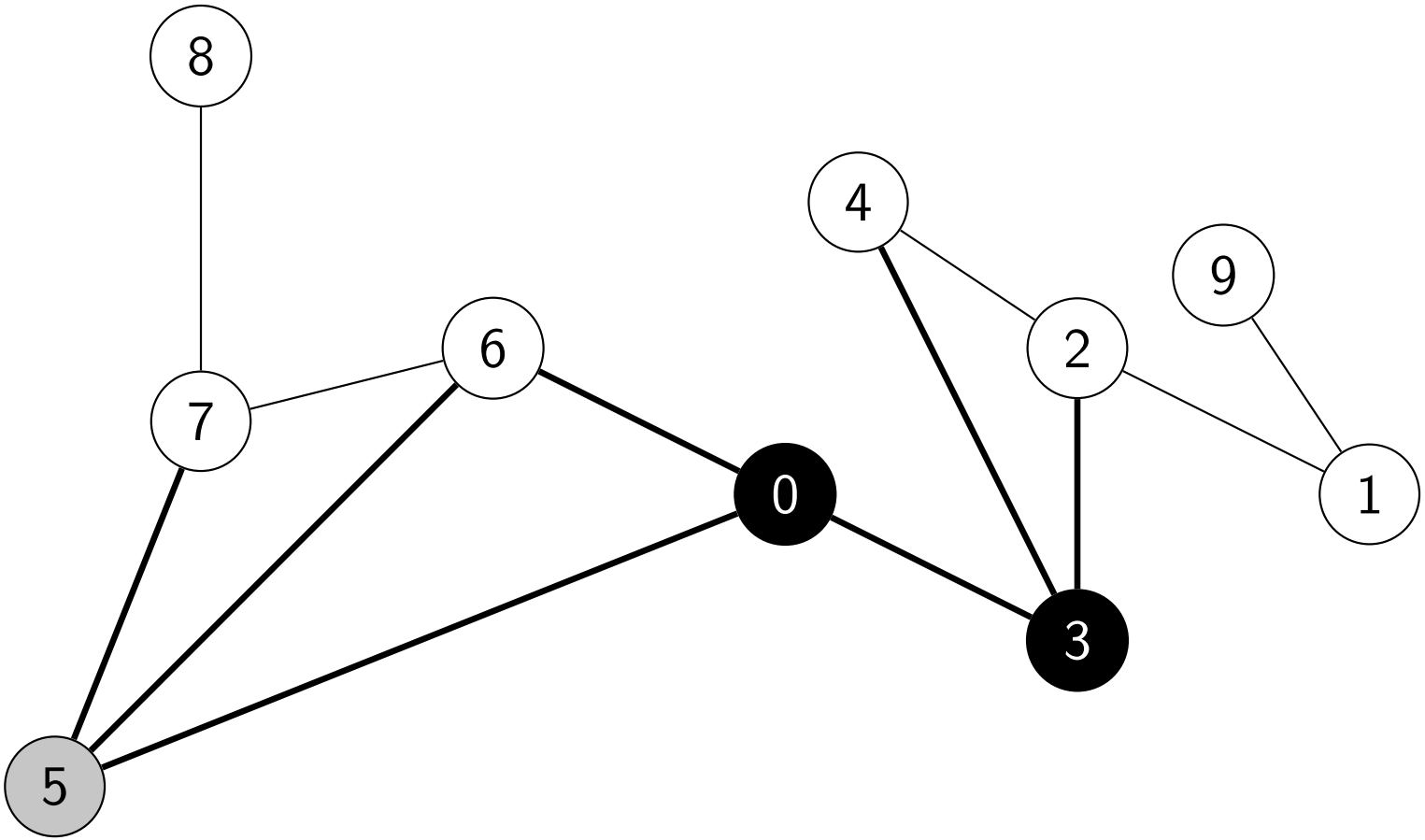


FIGURE 7 – On enfile les voisins à distance 2.

File : $7 \to 4 \to 2 \to 6$

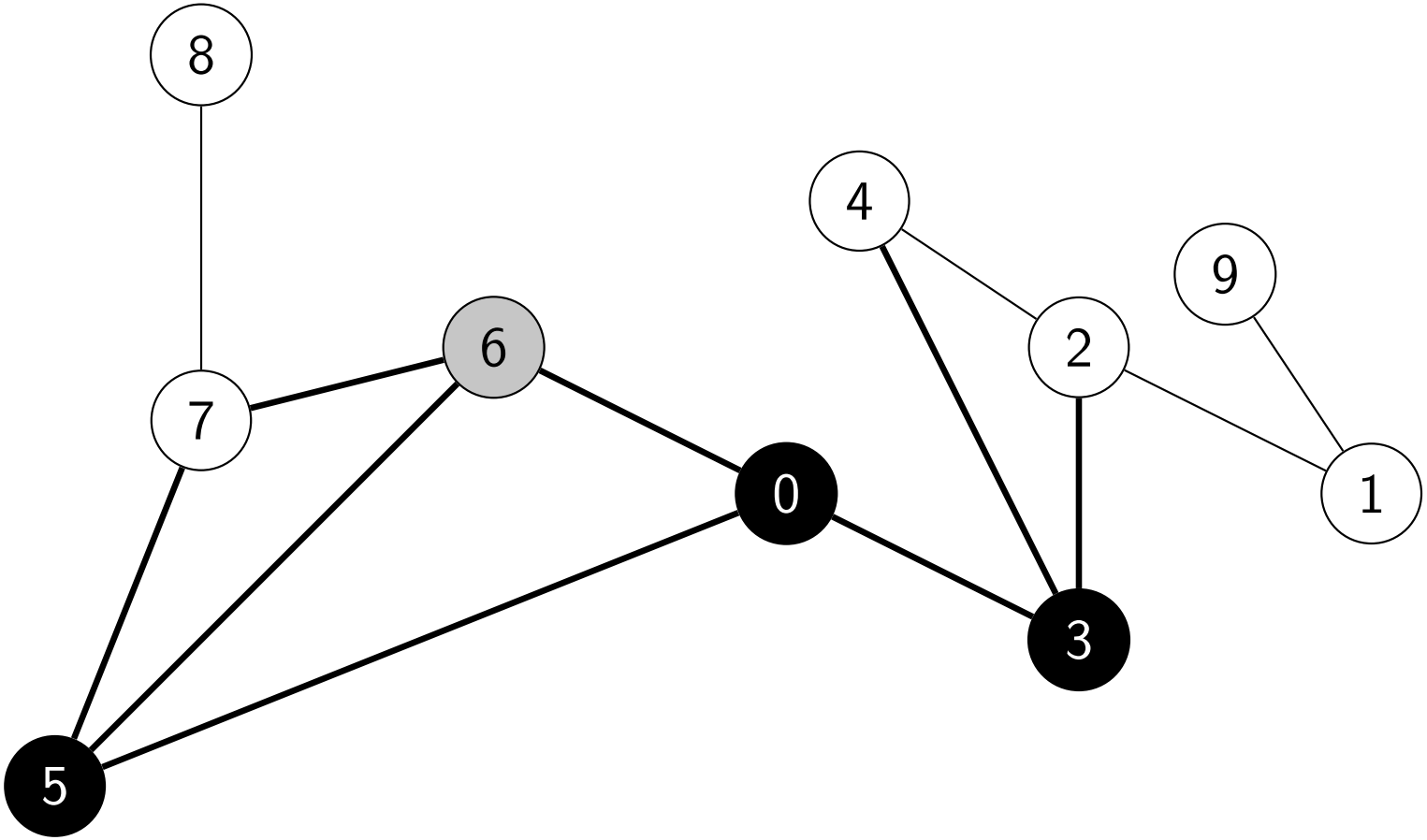


FIGURE 8 – On enfile les voisins à distance 2 : pas de nouveau sommet pour 6.

File : 7 → 4 → 2

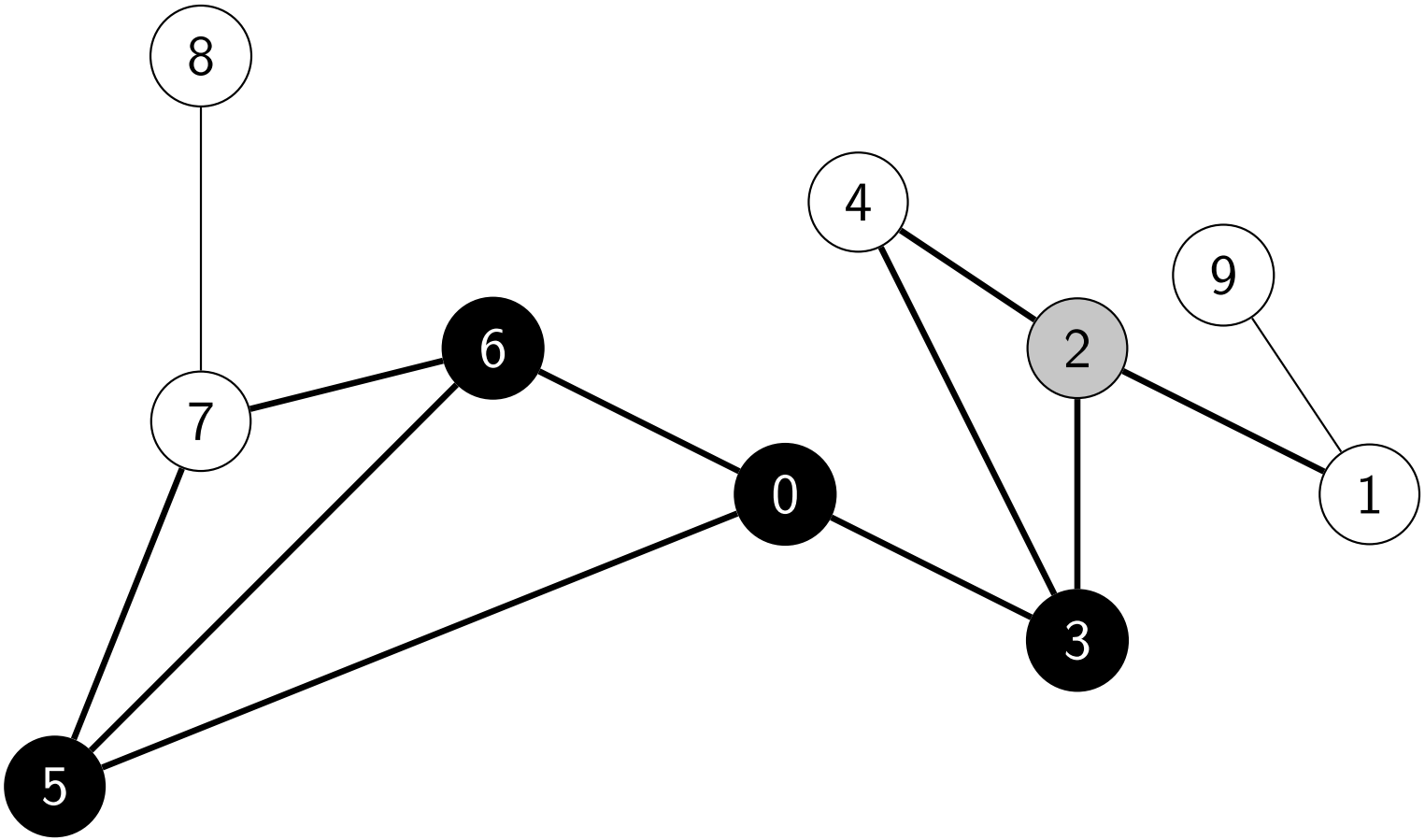


FIGURE 9 – On enfile les voisins à distance 3.

File : 1 → 7 → 4

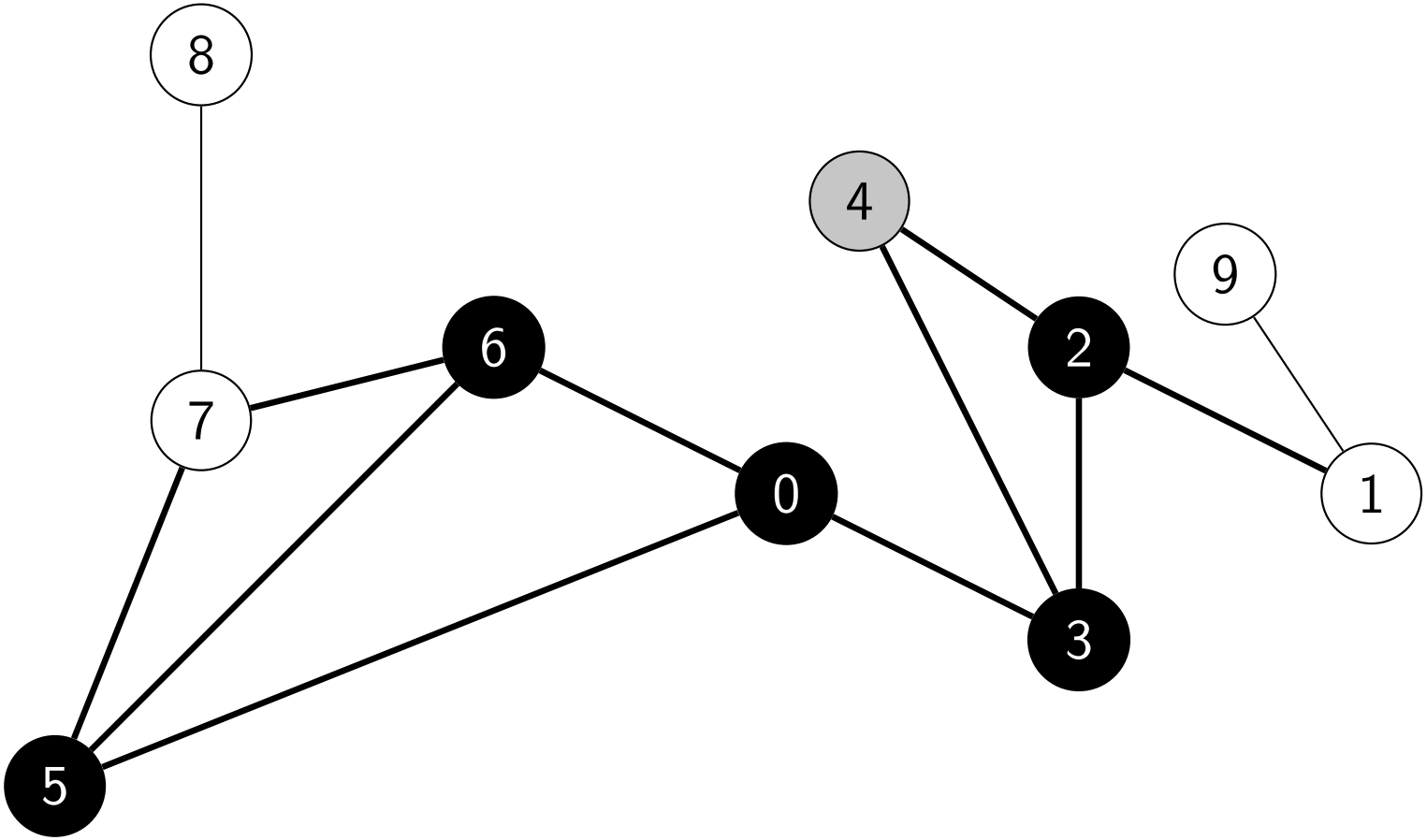


FIGURE 10 – On enfile les voisins à distance 3 : pas de nouveau sommet pour 4.

File : 1 → 7

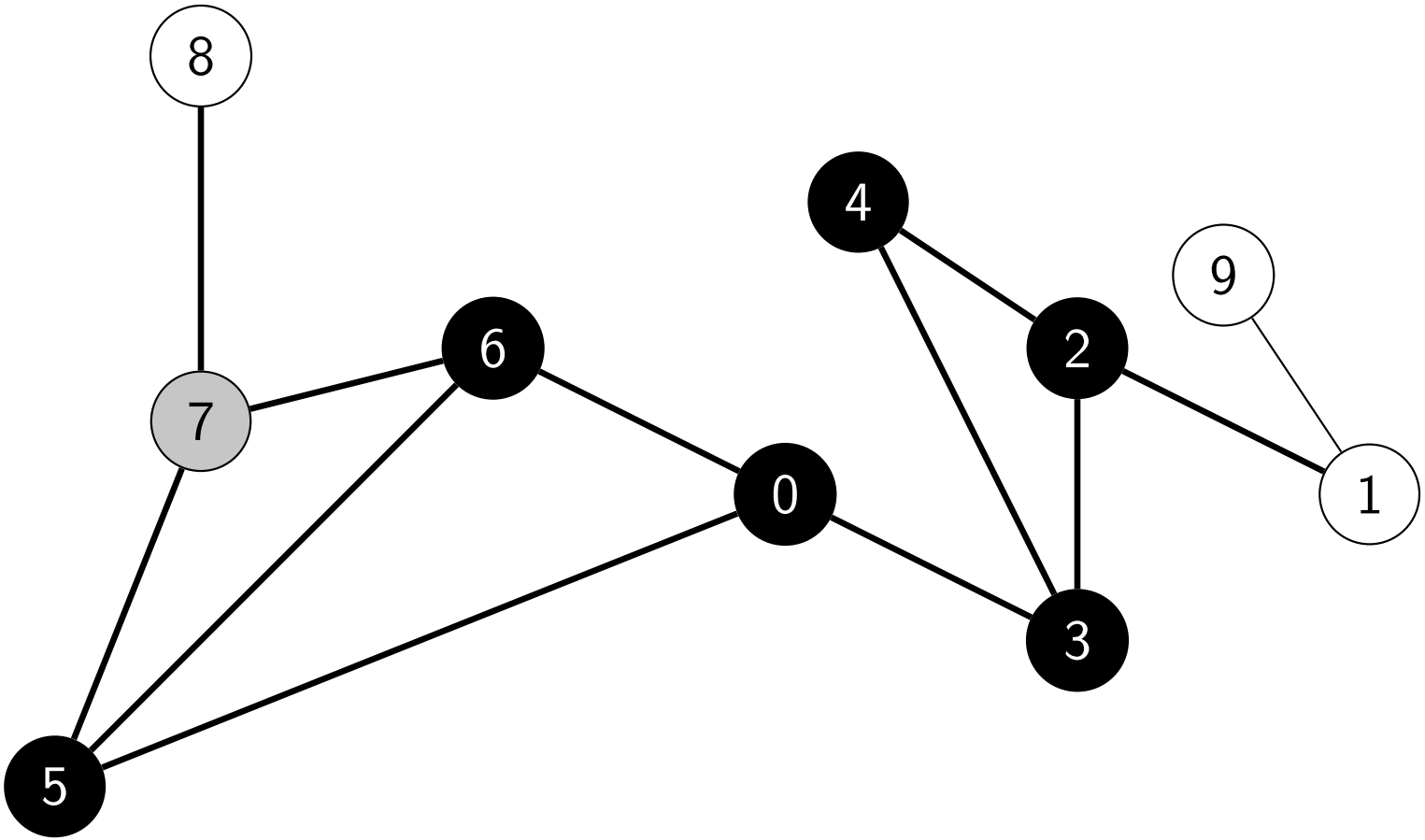


FIGURE 11 – On enfile les voisins à distance 3.

File : 8 → 1

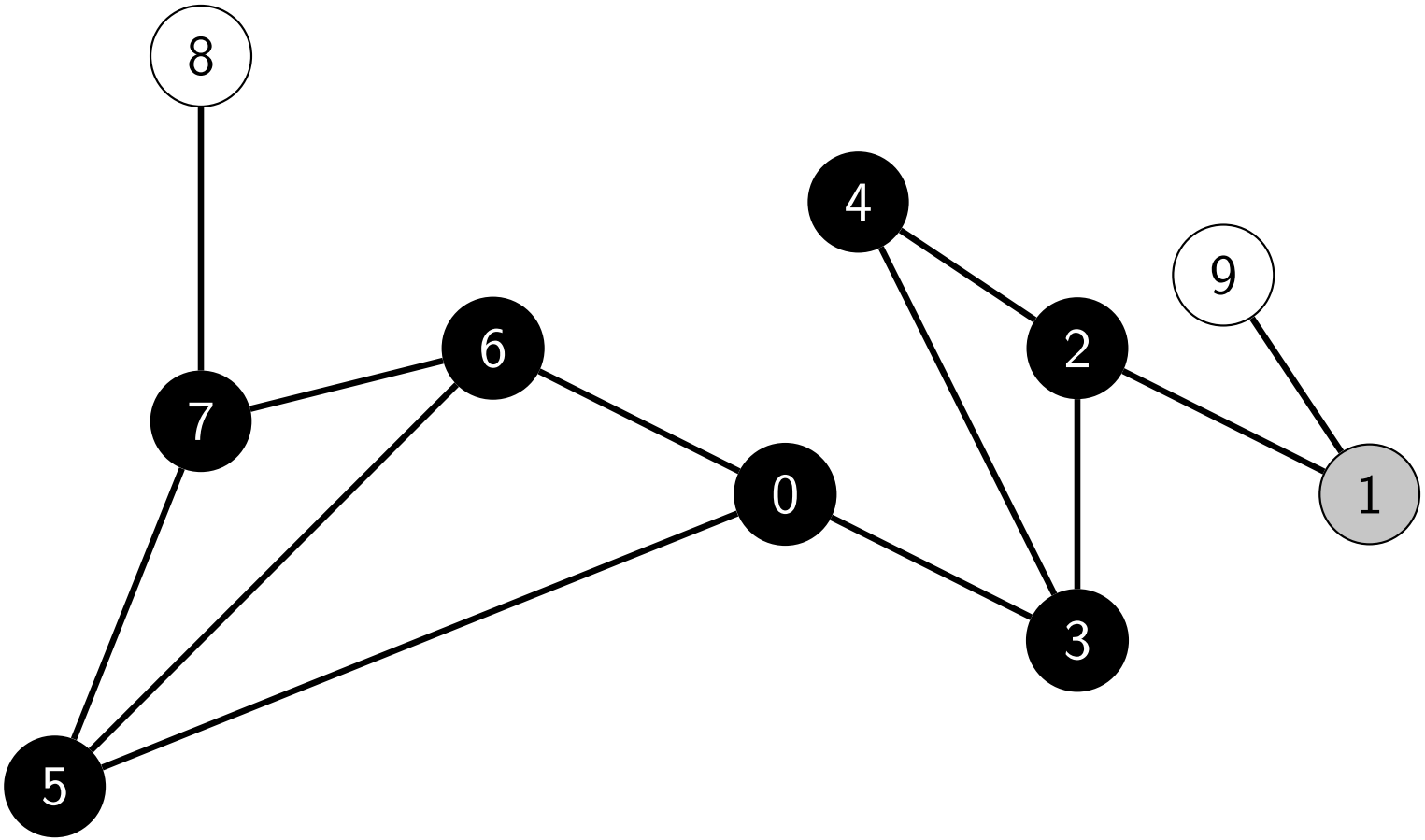


FIGURE 12 – On enfile les voisins à distance 4.

File : 9 → 8

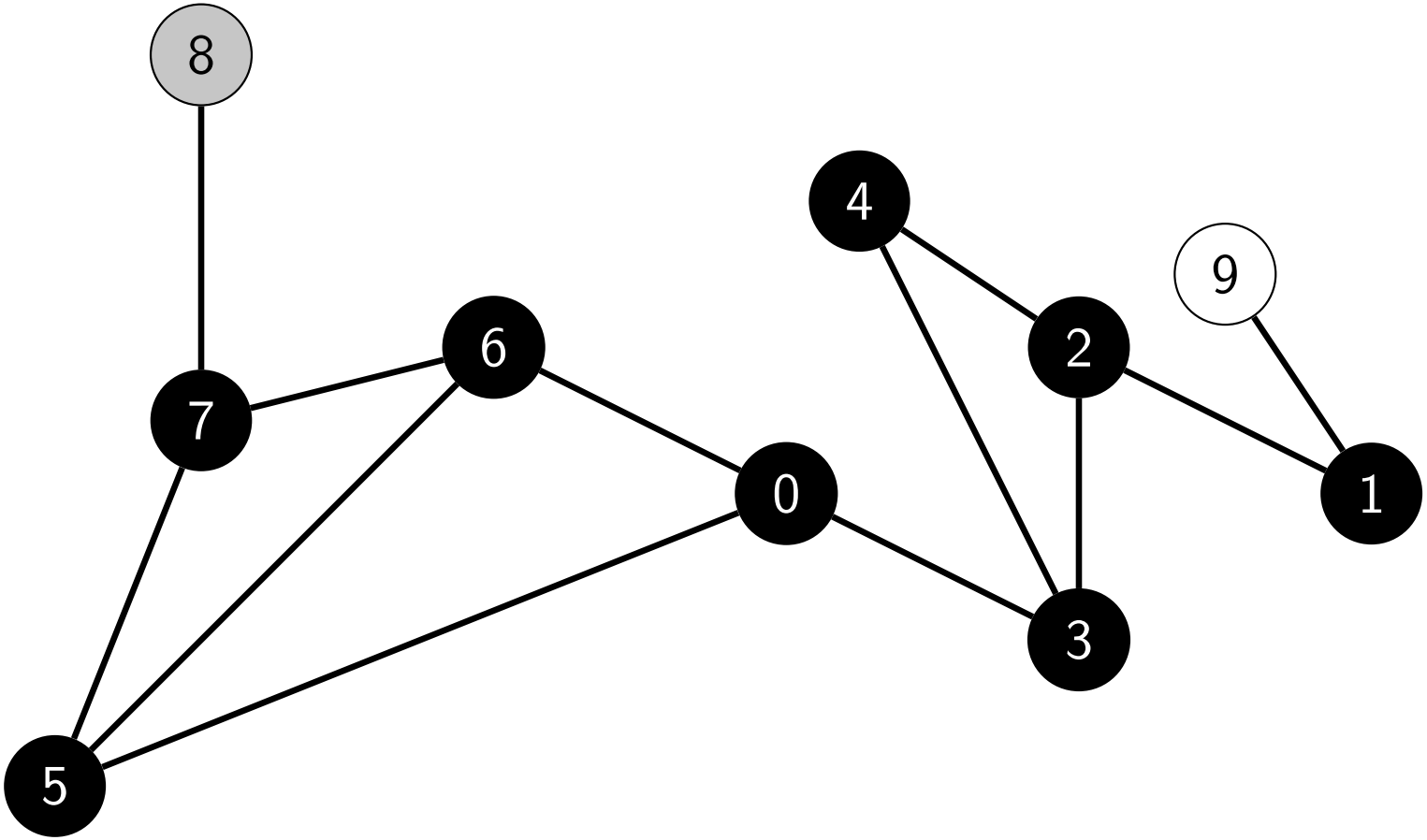


FIGURE 13 – On enfile les voisins à distance 4 : pas de nouveau sommet pour 8.

File : 9

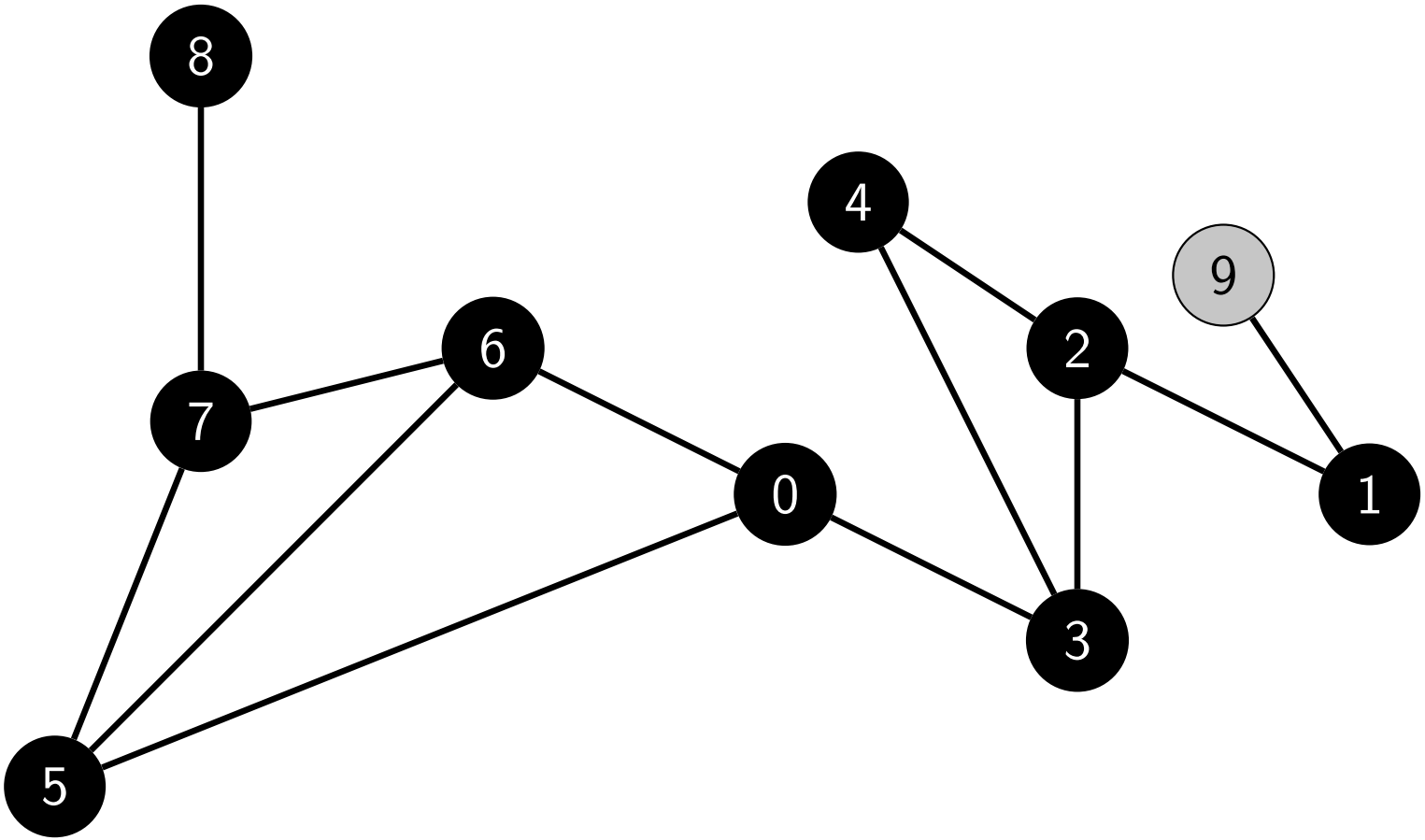


FIGURE 14 – Pas de sommet à distance 5.

File :

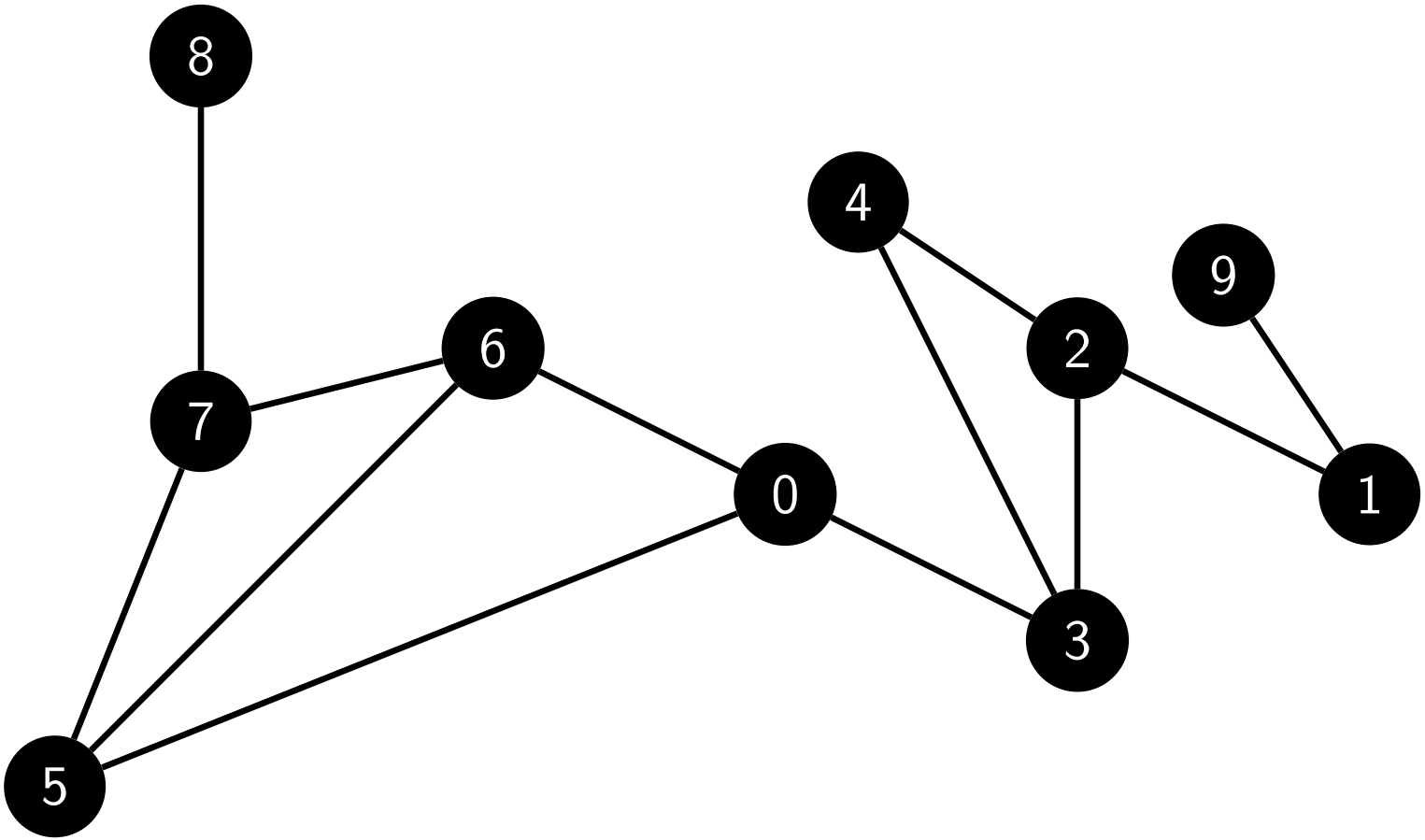


FIGURE 15 – Fin du parcours.

File :

Sommaire

1. Rappels

2. Parcours en profondeur

3. Parcours en largeur
3.1 Principe
3.2 Implémentation

Implémentation

Activité 6 :

1. Écrire la fonction `bfs(graphe: list)` $\rightarrow$ `list` qui effectue un parcours en largeur en utilisant une file. La fonction renverra la liste ordonnée des sommets parcourus.
2. Écrire la fonction `bfs_aretes(graphe: list)` $\rightarrow$ `list` qui renvoie les liste ordonnée des arêtes parcourues. La fonction colorera les sommets en :
   - `BLANC` au départ,
   - `GRIS` lors de l'empilement,
   - `NOIR` quand son parcours est terminé.

Parcours et structures linéaires
Rappels
Les structures linéaires
Représentation d'un graphe en mémoire
Parcours en profondeur
Parcours en largeur
Principe
Implémentation

Correction

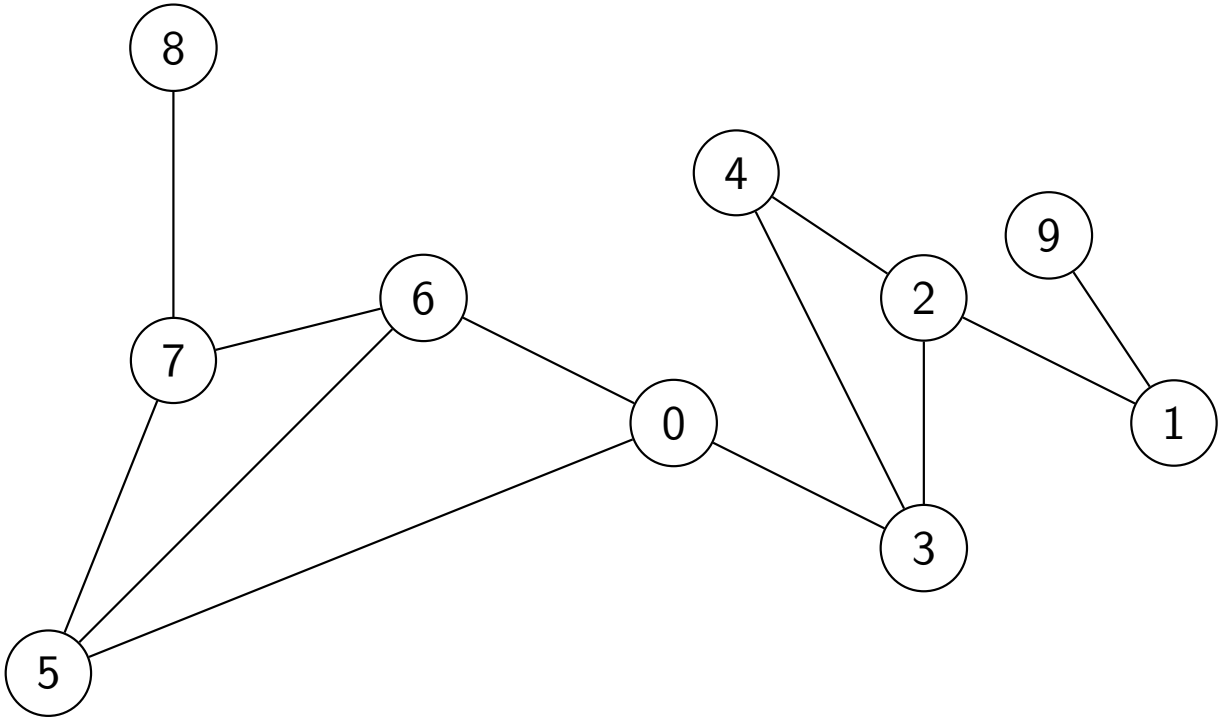
```
1  def bfs(graphe: list) -> list:
2      parcours = []
3      f = File()
4      f.enfiler(0)
5      while not f.est_vide():
6          en_cours = f.defiler()
7          if en_cours not in parcours:
8              parcours.append(en_cours)
9
10             for voisin in graphe[en_cours]:
11                 f.enfiler(voisin)
12     return parcours
```

```
1  >>> bfs(graphe)
2  [0, 3, 5, 6, 2, 4, 7, 1, 8, 9]
```

Code 3 – Appel de la fonction

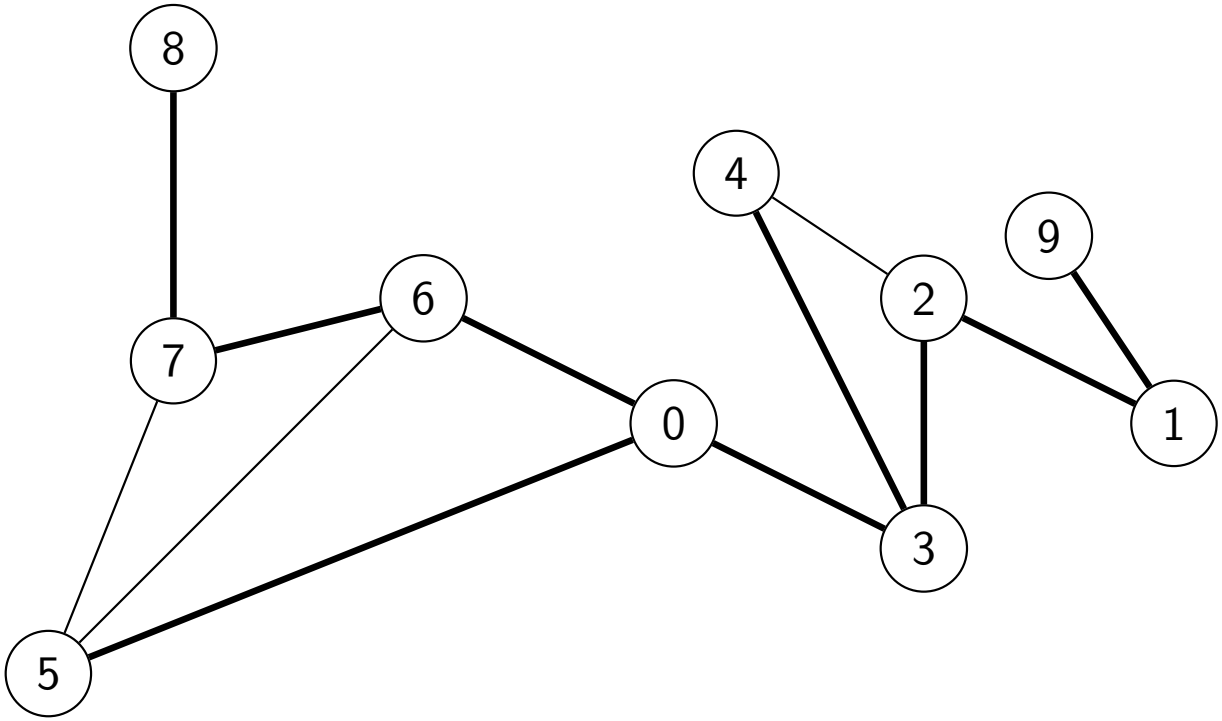
```python
def bfs_aretes(graphe: list) -> list:
    parcours = []
    etats = [BLANC for _ in range(len(graphe))]
    f = File()
    f.enfiler(0)
    etats[0]=GRIS
    while not f.est_vide():
        en_cours = f.defiler()
        for voisin in graphe[en_cours]:
            if etats[voisin] == BLANC:
                etats[voisin] = GRIS
                f.enfiler(voisin)

            # stockage arête parcourue
            parcours.append((en_cours, voisin))
        etats[en_cours] = NOIR
    return parcours
```




```
1 >>> bfs_aretes(graphe)
2 [(0, 3), (0, 5), (0, 6), (3, 0), (3, 2), (3, 4),
   (5, 0), (5, 6), (5, 7), (6, 0), (6, 5), (6, 7),
   (2, 1), (2, 3), (2, 4), (4, 2), (4, 3), (7, 5),
   (7, 6), (7, 8), (1, 2), (1, 9), (8, 7), (9, 1)]
```

Code 4 – Chaque sommet n'est enfilé qu'une seule fois. Chaque arête est parcourue deux fois (graphe non oirenté).




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

Observation

Si on stocke les arêtes seulement quand on enfile les voisins, elles ne sont pas toutes visitées.