Représentation d'un graphe en POO

Christophe Viroulaud

Terminale - NSI

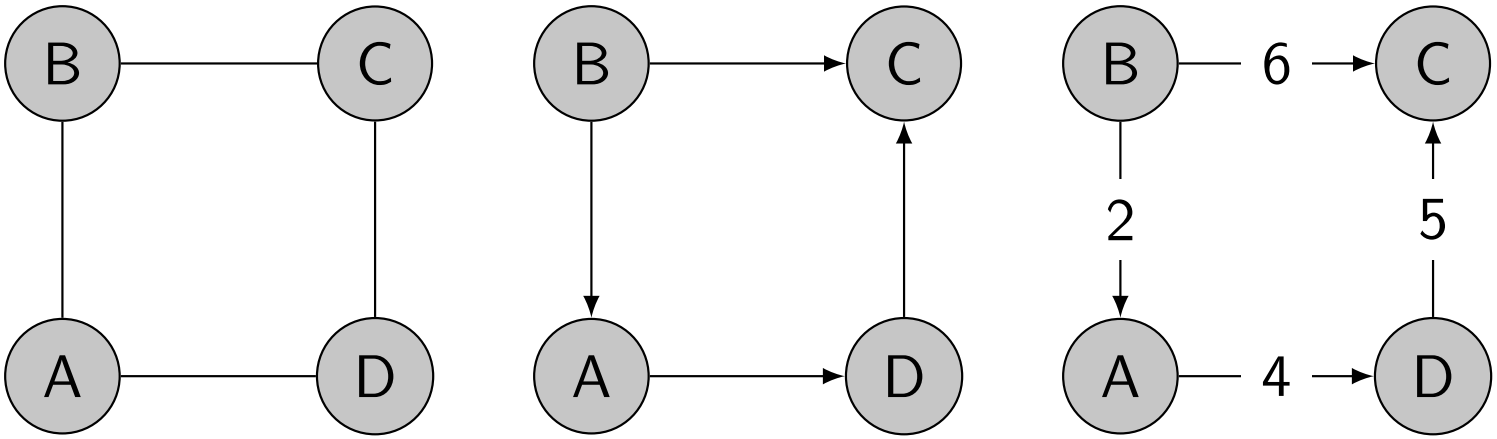
Algo 22

Les graphes sont des outils très utilisés en informatique. Il semble donc intéressant de disposer d'une bibliothèque fournissant les outils nécessaires à la création et la manipulation d'un graphe.

Construire une bibliothèque **`graphe`**.

Sommaire

Conception - Différents graphes



Observation

La bibliothèque **`graphe`** doit prendre en compte les différents cas de figures.

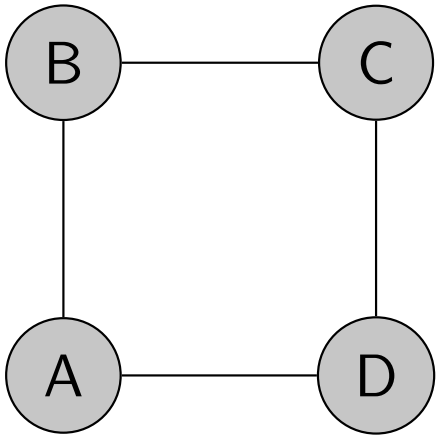
- Une arête non-orientée est équivalente à deux arêtes orientées de sens opposé.
- Un graphe non pondéré est équivalent à un graphe où toutes les pondérations valent 1.

Sommaire

Les ensembles

À retenir

Un ensemble est une collection non ordonnée sans éléments en double. https://tinyurl.com/set-pyt




```
1 voisins_B = { ("A", 1), ("C", 1) }
```

Code 1 – Ensemble des voisins de B

Remarque

```
1  # Crée un ensemble vide
2  e = set()
3
4  # Crée un dictionnaire vide
5  d = {}
```

Sommaire

Implémentation - Classe `Graphe`

Activité 1 :

1. Créer le fichier `biblio_graphe.py`
2. Construire la classe `Graphe` et son constructeur qui initialisera :
   - un attribut booléen `oriente` qui sera initialisé par une valeur booléenne passée en paramètre.
   - un dictionnaire vide nommé `sommets`.

Avant de regarder la correction

- Prendre le temps de réfléchir,
- Analyser les messages d'erreur,
- Demander au professeur.

Correction

```
1  class Graphe:
2      def __init__(self, oriente: bool):
3          self.sommets = {}
4          self.oriente = oriente
```

Activité 2 :

1. Écrire la méthode **`ajouter_sommet(self, s: str)`** $\rightarrow$ **`None`** qui ajoute un nouveau sommet dans le dictionnaire **`sommets`** s'il n'est pas déjà présent. Un ensemble vide sera associé au sommet.
2. Écrire la méthode **`ajouter_arete(self, s1: str, s2: str, d: int = 1)`** $\rightarrow$ **`None`** qui crée une arête, éventuellement orientée de **`s1`** vers **`s2`**, de pondération **`d`**. Si la pondération n'est pas précisée la valeur `1` est utilisée par défaut. Si les sommets n'existent pas, la méthode devra les créer préalablement.

Avant de regarder la correction

- Prendre le temps de réfléchir,
- Analyser les messages d'erreur,
- Demander au professeur.

Correction

```
1  def ajouter_sommet(self, s: str) -> None:
2      if not(s in self.sommets):
3          self.sommets[s] = set()
4
5  def ajouter_arete(self, s1: str, s2: str, d:
       int = 1) -> None:
6      # ajout éventuel des sommets
7      self.ajouter_sommet(s1)
8      self.ajouter_sommet(s2)
9      # création arête
10     self.sommets[s1].add((s2, d))
11     if not self.oriente:
12         self.sommets[s2].add((s1, d))
```

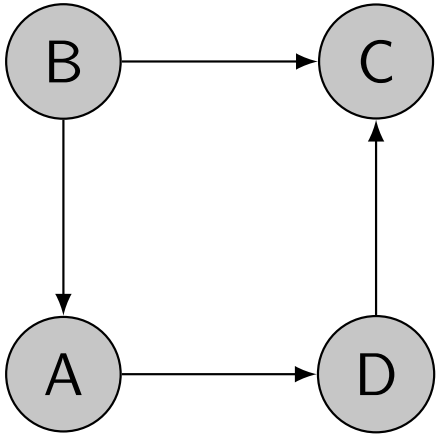


FIGURE 1 – Exemple

Activité 3 :

1. Créer un fichier **`exemple.py`**
2. Importer la bibliothèque et créer une instance de **`Graphe`** représentant le graphe (figure 1).
3. Dans la classe **`Graphe`** créer la méthode **`affiche_voisins(self, s: str)`** $\rightarrow$ **`None`** qui affiche dans la console les voisins de **`s`** ainsi que la distance entre les 2 sommets.

Avant de regarder la correction

- Prendre le temps de réfléchir,
- Analyser les messages d'erreur,
- Demander au professeur.

Correction

```
1 exemple = Graphe(True)
2 exemple.ajouter_arete("A", "D")
3 exemple.ajouter_arete("D", "C")
4 exemple.ajouter_arete("B", "A")
5 exemple.ajouter_arete("B", "C")
```

Code 2 – Instanciation

```
1 def affiche_voisins(self, s: str) -> None:
2     for v in self.sommets[s]:
3         print(f"{s} --> {v[0]} : {v[1]}")
```

Code 3 – Méthode d'affichage

```
1 >>> exemple.affiche_voisins("B")
2 B --> A : 1
3 B --> C : 1
```

Code 4 – Appel de la méthode

Sommaire

Méthodes de parcours

Activité 4 :

1. Dans le même dossier que la bibliothèque, placer le fichier **`structures.py`** construit dans les cours précédents.
2. Écrire la méthode itérative **`dfs_it(self, s: str)`** $\rightarrow$ **`list`** qui renvoie la liste des sommets visités depuis **`s`** avec un parcours en profondeur.
3. **Pour les plus avancés :** Écrire la méthode récursive **`dfs_rec(self, s: str, parcours: list)`** $\rightarrow$ **None**.

Avant de regarder la correction

- Prendre le temps de réfléchir,
- Analyser les messages d'erreur,
- Demander au professeur.

Correction

```python
def dfs_it(self, s: str) -> list:
    parcours = []
    p = Pile()
    p.empiler(s)
    while not p.est_vide():
        en_cours = p.depiler()
        if en_cours not in parcours:
            parcours.append(en_cours)

            for voisin in self.sommets[en_cours]:
                p.empiler(voisin[0])

    return parcours
```

```
>>> exemple.dfs_it("A")
['A', 'D', 'C']
```

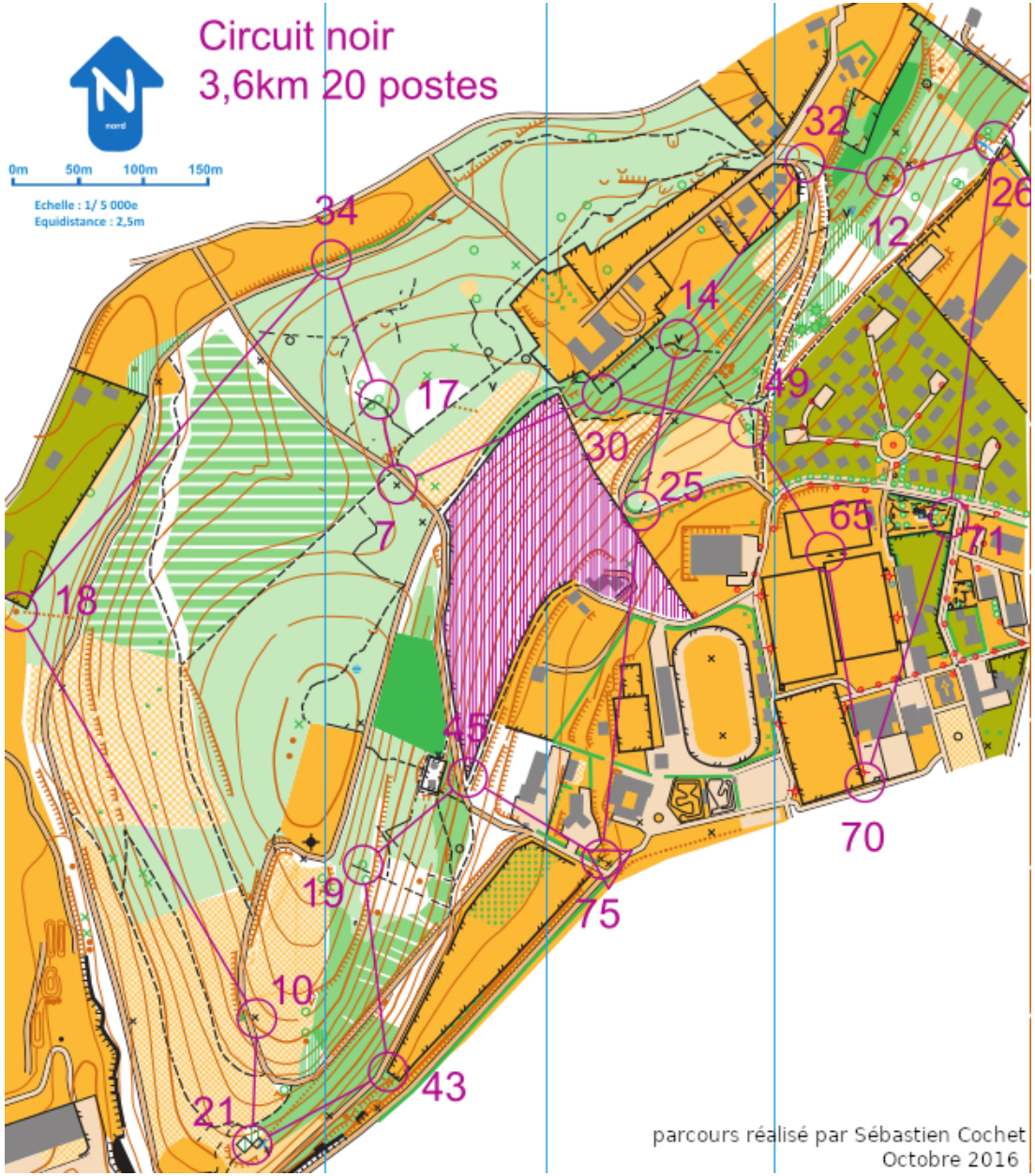
```python
def dfs_rec(self, s: str, parcours: list) -> None:
    """
    version récursive du parcours en profondeur

    Args:
        s (str): départ
        parcours (list): sommets traversés
    """
    if s not in parcours:
        parcours.append(s)
        for voisin in self.sommets[s]:
            self.dfs_rec(voisin[0], parcours)
```

```
>>> parcours_rec = []
>>> exemple.dfs_rec("A", parcours_rec)
>>> parcours_rec
['A', 'D', 'C']
```

Sommaire

Application



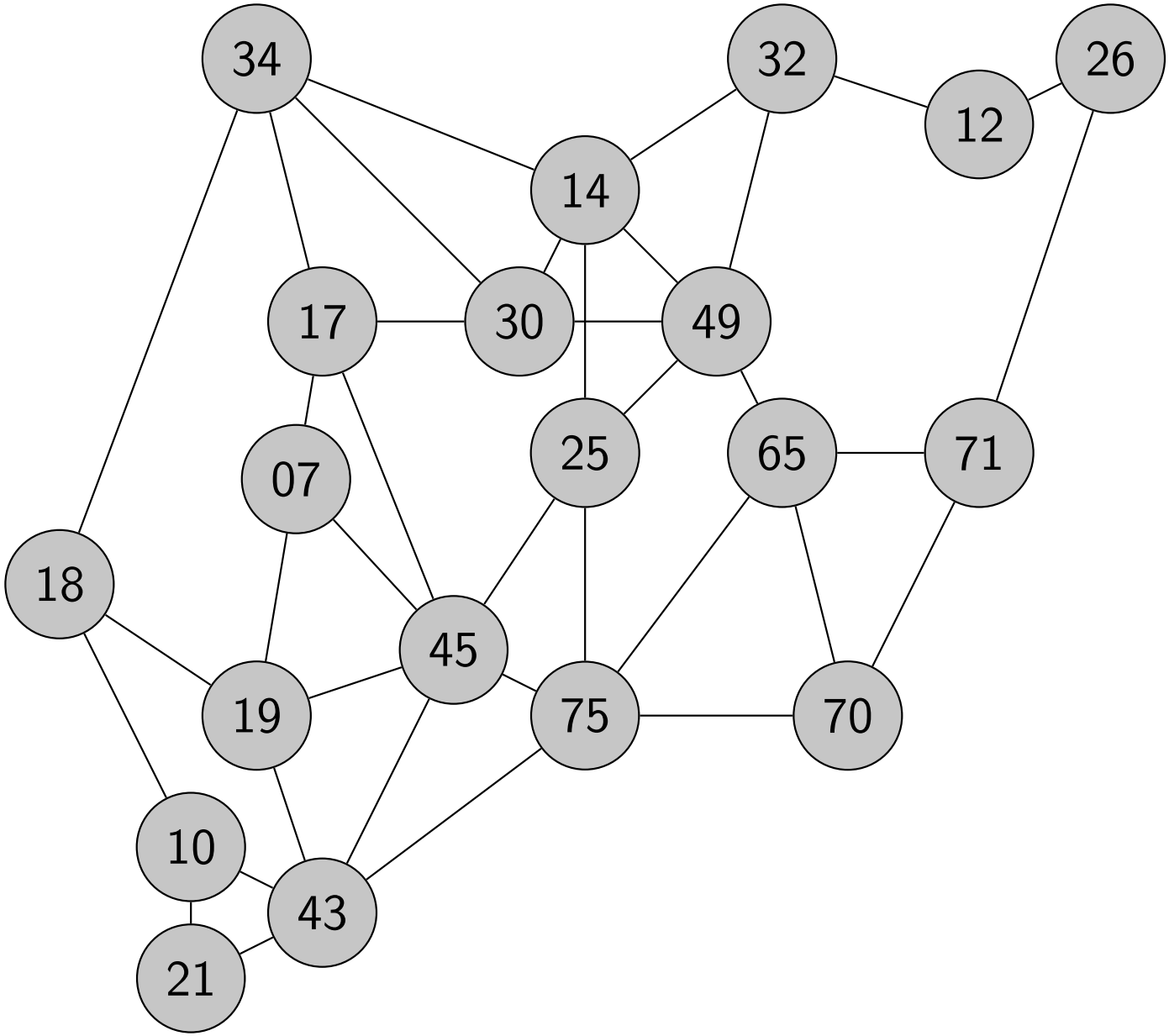


FIGURE 2 – Graphe représentatif de la carte de CO

Activité 5 :

1. Placer le fichier **`parcours_noir.json`** dans le même dossier que la bibliothèque.
2. Créer un fichier **`parcours_co.py`**
3. Construire une instance de **`Graphe`** représentative de la carte de CO.

Avant de regarder la correction

- Prendre le temps de réfléchir,
- Analyser les messages d'erreur,
- Demander au professeur.

Correction

```python
import json
from biblio_graphe import Graphe

# chargement du json
f = open("parcours_noir.json")
list_graphe = json.load(f)

graphe = Graphe(False)
# parcours du graphe
for couple in list_graphe:
    sommet = couple["sommet"]
    adjacents = couple["adjacents"]
    for voisin in adjacents:
        # les sommets sont ajoutés automatiquement
        graphe.ajouter_arete(sommet, voisin)

f.close()
```

Sommaire

Pour aller plus loin

Considérons un graphe non orienté et non pondéré. Un parcours en largeur permet de déterminer le plus court chemin entre deux sommets. En effet, le parcours *s'éloigne* de l'origine niveau après niveau. Il suffit alors de maintenir un dictionnaire des sommets visités associés à leur prédécesseur pour retrouver le chemin le plus court partant de l'origine jusqu'à n'importe quel sommet.

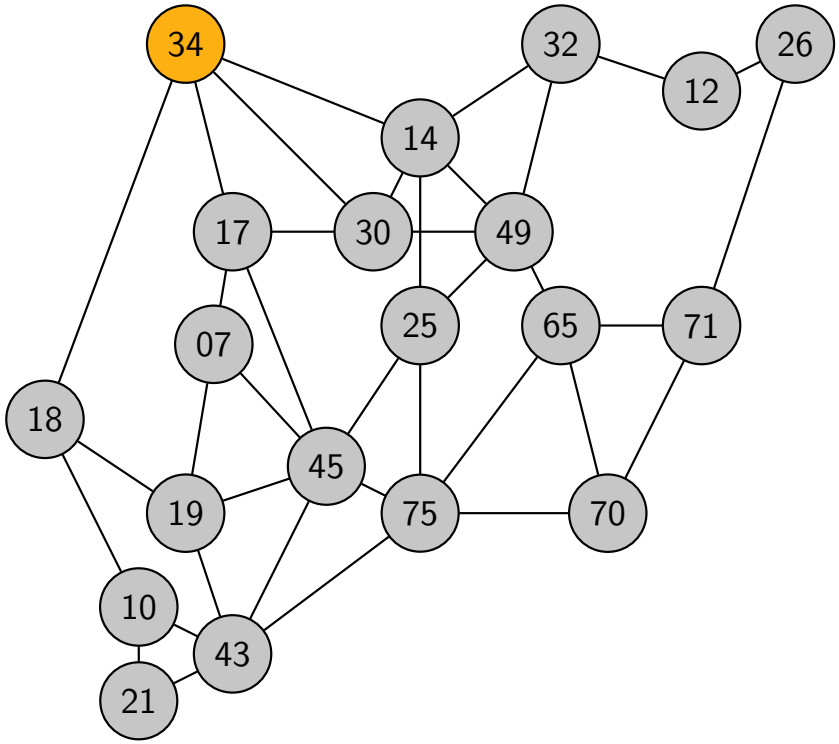


FIGURE 3 – Plus court chemin entre 34 et 19.

- 34 : niveau 0

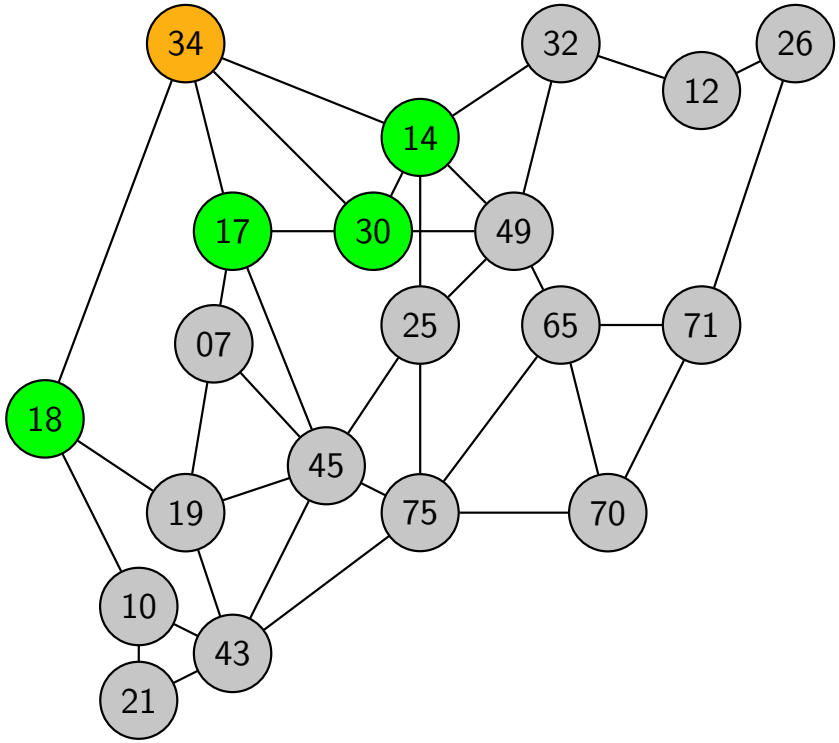


FIGURE 4 – Plus court chemin entre 34 et 19.

- 34 : niveau 0
- 14 - 30 - 17 - 18 : niveau 1

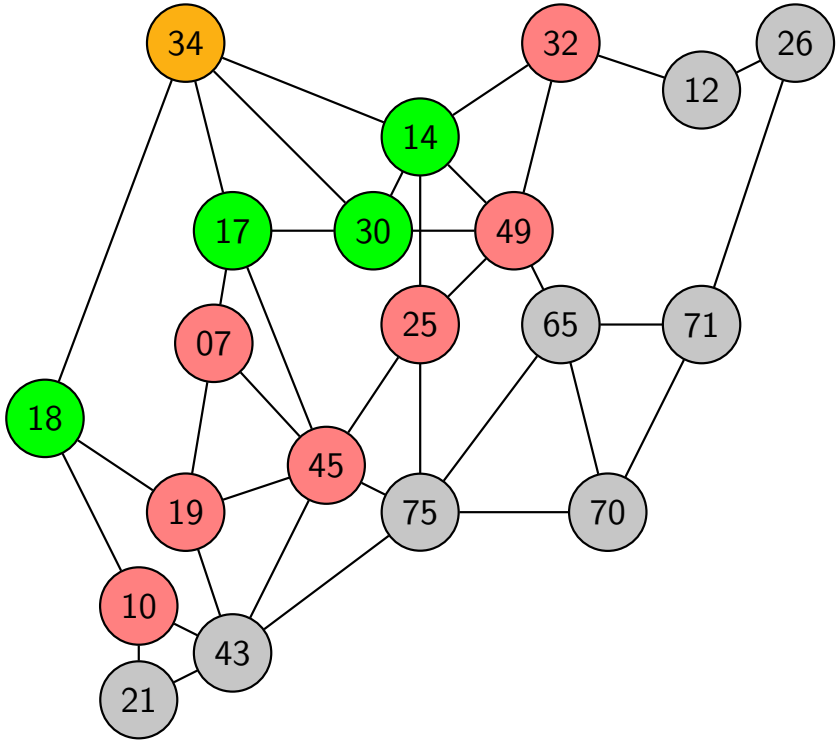


FIGURE 5 – Plus court chemin entre 34 et 19.

- 34 : niveau 0
- 14 - 30 - 17 - 18 : niveau 1
- 32 - 49 - 25 - 45 - 07 - 19 - 10 : niveau 2