Représentation d'un graphe en Python

Il existe de multiples façons de représenter un graphe en machine, selon la nature du graphe mais aussi selon des opérations et des algorithmes à effectuer sur ce graphe. Quelle que soit la représentation, on souhaite proposer des opérations de deux sortes :

. d'une part des opérations de construction de graphes, comme l'obtention d'un graphe vide, l'ajout de sommets ou d'arcs, etc ;

. d'autre part des opérations de parcours d'un graphe, comme parcourir tous les sommets, tous les arcs ou encore tous les arcs issus d'un sommet donné.

A priori, on ne souhaite pas fixer le type des sommets d'un graphe. Ce pourrait être des entiers, des chaînes de caractères ou encore des objets.

Cependant, nous allons commencer par une représentation où les sommets sont des entiers, avant de proposer une représentation plus souple où les sommets peuvent être d'un type quelconque.

On se focalise pour l'instant sur des graphes orientés et on expliquera, à la fin de cette section, comment représenter des graphes non orientés.

Matrice d'adjacence

Dans cette première représentation, les sommets du graphe sont supposés être les entiers 0,1,2,…,

N-1 pour un certain entier N, qui se trouve donc être le nombre total de sommets. On peut alors représenter le graphe par une matrice adj de booléens de taille $N \times N$, c'est-à-dire un tableau de N tableaux de booléens de taille N, où le booléen adj$[i][j]$ indique la présence d'un arc entre les sommets $i \; et \; j$.On appelle cela une matrice d'adjacence.

Pour construire un certain graphe, on peut commencer par construire une telle matrice où tous les booléens sont initialisés à False, par exemple de la manière suivante :

```
Adj = [ [False] * N for _ in range(N)]
```

Ensuite, on peut ajouter des arcs au graphe, en donnant la valeur True à certains éléments de cette matrice. Ainsi, on peut ajouter un arc entre les sommets 2 et 7 avec une simple affectation :

```
Adj[2][7] = True
```

Et ainsi de suite.

Une telle représentation a le mérite d'être relativement simple. En particulier, on peut parcourir tous les sommets du graphe avec une simple boucle :

```
for  s  in range(N) :
```

Voici un programme qui vous est proposé:

Class Graphe :

" " " un graphe représenté par une matrice d'adjacence, où les sommets sont les entiers 0,1,...,n-1" " "

```
def __init__(self , n) :
    self.n = n
    self.adj = [[False]* n for _ in range(n)]
def ajouter_arc(self , s1 , s2):
    self.adj[s1][s2] = True
def arc(self , s1 , s2):
    return self.adj[s1][s2]
def voisins(self , s):
    v = □
    for i in range(self.n):
        if self.adj[s][i]:
            v.append(i)
    return v
```

Commenter les sripts .

De même, on peut parcourir tous les voisins du sommet s avec une boucle for et un test :

```
for v in range(N) :
```

in adj[s][v]:

Ce programme encapsule une telle matrice d'adjacence dans une classe Graphe. Le constructeur de cette classe prend en argument le nombre de sommets et alloue la matrice. Une méthode

ajouter_arc permet d'ajouter un arc au graphe. Ainsi, on peut écrire :