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 c’est-à-dire un tableau de N tableaux de booléens de taille N, où le booléen adj[ indique la présence d’un arc entre les sommets 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 :

g = Graphe(4)

g.ajouter\_arc(0 , 1)

g.ajouter\_arc(0 , 3)

g.ajouter\_arc(1 , 2) 1

g.ajouter\_arc(3 , 1) 0 2

3

Pour construire le graphe représenté à droite. Une méthode arc permet de tester la présence d’un arc entre deux sommets et une méthode voisins renvoie l’ensemble des voisins d’un sommet donné, sous forme d’un tableau.

Nous traiterons en TP une méthode afficher à cette classe.

Efficacité : La matrice d’adjacence est indéniablement simple à mettre en œuvre, mais elle a néanmoins quelques défauts. D’une part, elle occupe un espace mémoire proportionnel à nécessite une matrice d’un million de booléens, ce qui représente déjà quelques mégaoctets, et ce, même si le graphe contient très peu d’arcs. D’autre part, parcourir tous les voisins d’un sommet donné exige de parcourir toute une ligne de matrice, c’est-à-dire

Y avoir très peu de voisins. Enfin, elle limite les sommets à des entiers, qui plus sont consécutifs et d’un nombre connu à l’avance. Dans la section suivante, nous présentons une autre façon de représenter un graphe, qui s’affranchit de tous ces défauts.

Dictionnaire d’adjacence :

Dans cette nouvelle représentation, un graphe est un dictionnaire, qui associe à chaque sommet l’ensemble de ses voisins. On appelle cela un dictionnaire d’adjacence. La première conséquence est que les sommets ne sont pas limités à des entiers et qu’il n’est pas nécessaire de les connaître tous a priori. En effet, il suffit d’ajouter une nouvelle entrée dans le dictionnaire pour ajouter un nouveau sommet au graphe. L’ensemble des sommets du graphe est exactement l’ensemble des clés du dictionnaire. En particulier, on peut parcourir tous les sommets d’un graphe stocké dans la variable graphe avec la boucle suivante :

for s in range :

…

Les voisins du sommets s sont donnés par l’exemple graphe[s]. On peut donc les parcourir avec la boucle suivante :

for v in graphe[s]:

La deuxième conséquence de cette nouvelle représentation est que ce parcours est maintenant d’une complexité directement proportionnelle au nombre de voisins de s , indépendamment du nombre total de sommets du graphe.

Le programme suivant encapsule un tel dictionnaire d’adjacence dans une classe Graphe. Le constructeur de cette classe alloue un dictionnaire vide.

Une méthode ajouter\_sommet permet d’ajouter un nouveau sommet et une méthode ajouter\_arc permet d’ajouter un arc. Ainsi, on peut écrire

g = Graphe()

g.ajouter\_arc(0 , 1)

g.ajouter\_arc(0 , 3 )

g.ajouter\_arc(3 , 1)

Voici un programme sur le Graphe représenté par un dictionnaire d’adjacence :

Class Graphe :

‘’ ‘’ ‘’ un graphe comme un dictionnaire d’adjacence ‘’ ‘’ ‘’

def \_ \_init\_ \_(self) :

self.adj = {}

def ajouter\_sommet(self , s):

if s not in self.adj:

self.adj[s] = set()

def ajouter\_arc(self , s1 , s2):

self.ajouter\_sommet(s1)

self.ajouter\_sommet(s2)

self.adj[s1].add(s2)

def arc(self , s1 , s2):

return s2 in self.adj[s1]

def sommets(self):

return list(self.adj)

def voisins(self , s):

return self.adj[s]

0 2

3