Exercice 1

November 1, 2019

1 Exercice 1

1.1 Graphes

Un `graphe` est une data structure dans laquelle les éléments sont reliés les uns aux autres de manière arbitraire.

Les éléments du `graphe` sont les `noeuds` et les `arrêtes`, les `noeuds` contiennent les valeurs et sont reliés les uns aux autres par les `arrêtes` (notez que les arrêtes elles-mêmes peuvent contenir des informations.)

1.2 Utilité:

Les `graphes` sont très utilisés en sciences forensiques car ils permettent de facilement faire le rapprochement entre différents individus ou différents évènements. Ils peuvent être utilisés pour identifier les réseaux de criminels, identifier des fraudes sur des systèmes complexes, et même résoudre des cas de meurtres non résolus en les reliant à des tueurs en séries.

De plus les `graphes` sont à la base de l'`intelligence artificielle` puisque les `réseaux de neurons`, qui émulent la façon dont fonctionnent les cerveaux sont en fait des `graphes`.

1.3 Illustration:

1.3.1 Noeuds:

1.3.2 Arrêtes:

1.3.3 Graphe complet:

1.4 Graphes orientés:

Certains graphes peuvent avoir des `orientations`, c'est-à-dire que leurs `arrêtes` ne peuvent aller que dans un sens, c'est le cas pour les graphes relationnels. On note alors les directions par des flêches. Par exemple un graphe dans lequel on souhaiterait représenter une famille où Jean est le père et Sylvie la fille, leur relation serait la suivante: `(Jean)-[Père]->(Sylvie)`

Dans l'image suivante nous avons un graphe relationnel où les différentes entités peuvent travailler ou aimer d'autres entités:

1.5 Cartes

Une carte reliant différentes villes peut être exprimée sous forme de graphe. Par exemple, le graphe suivant montre les différentes distances entre Lyon, Cologne, Berlin, Hamburg et Paris

Nous aurions aussi pu représenter ce graphe avec une matrice 5*5:

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 700 & 0 & 0 & 0 & 550 \\ 1300 & 500 & 0 & 0 & 900 \\ 1100 & 850 & 400 & 0 & 1000 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Où chaque élément

$$M_{i,j} = distance(i, j)$$

Si une distance est marquée comme 0, il n'y a pas de chemin possible entre les deux villes

Tel que:

```
Lyon = 1
Cologne = 2
Hamburg = 3
Berlin = 4
Paris = 5
```

1.6 Créer un graphe de distances:

Pour créer un graphe, il suffit d'appeler le constructeur `Graph(matrix)` et de lui donner une matrice de distances en argument

1.6.1 Exemple:

```
graphe = Graph([[1, 2],[3, 4]])
```

Par défaut les éléments du `graphe` ont comme valeur les lettres en majuscule dans l'ordre (l'élément 1 vaut `"A"`, l'élément 2 vaut `"B"`, etc.)

1.7 Exercice:

Créez un graphe qui réplique le graphe de villes précédent

[1]:
```
from Graph import Graph

graphe = Graph([
    [0, 0, 0, 0, 0],
    [700, 0, 0, 0, 550],
    [1300, 500, 0, 0, 900],
    [1100, 850, 400, 0, 1000],
    [0, 0, 0, 0, 0]
])
```

[2]:
```
from assertion import graph_eq
graph_eq(graphe)
```

```
"Bonne réponse!"
```

1.8 Récupérer une node dans un graphe:

Pour récupérer un noeud dans un graphe, utilisez la fonction `get_node(str)` qui prend en argument la valeur de l'arrête à chercher dans le graphe.

Par exemple, pour récupérer la valeur `A` dans `graphe`, utilisez `graphe.get_node("A")`

Chaque `Node` contient une liste d'arrêtes que vous pouvez récupérer avec `node.relationships`. Et chaque `arrête` contient l'élément d'origine `relationship._from` et l'élément vers laquelle elle pointe `relationship.to`.

Donc pour récupérer une node, et afficher la valeur de la première node à laquelle elle est ratachée, faites:

```
node = graphe.get_node("B")
relationships = node.relationships
first_rel = relationships[0]
first_neighbour = firs_rel.to
print(first_neighbour.value)
```

Pour accéder à la distance entre deux nodes grâce à une `arrête`, faites `relationship.value` et pour voir le nom d'un `noeud`, faites `node.value`.

1.9 Exercice:

Toujours avec votre graphe précédent, récupérez la `Node "D"` et affichez la valeur de toutes les nodes à laquelle elle est connectée, ainsi que la distance.

L'output devrait être

```
A 1100
B 850
C 400
E 1000
```

[2]:
```
# Votre code ici
node = graphe.get_node("D")
relationships = node.relationships

for rel in relationships:
    print(rel.to.value, rel.value)
```

```
A 1100
B 850
C 400
E 1000
```

1.10 Exercice

Implémentez l'algorithme de `Djikstra` pour trouver le chemin le plus proche entre deux `noeuds`. Votre fonction devrait retourner un tuple contenant en première position `0`, la distance et en seconde

position 1, le chemin parcouru.

[4]:
```python
from math import inf

def dijkstra(origin, destination, visited = None):
    if visited is None:
        visited = set()
    # VOTRE CODE ICI
    if origin.value == destination:
        return (0, origin.value)
    distance = inf
    path = origin.value
    visited.add(origin.value)
    for relationship in origin.relationships:
        neighbour = relationship.to
        if neighbour.value not in visited:
            distance_temp, path_temp = dijkstra(neighbour, destination, visited)
            total_distance = distance_temp + relationship.value
            if total_distance  < distance:
                distance = total_distance
                path = origin.value + path_temp
    return (distance, path)
```

[5]:
```python
from Graph import Graph
from assertion import assert_dijkstra

vertices = [
    [0, 3, 0, 5, 10, 9, 2, 40],
    [1, 0, 10, 3, 2, 1, 29, 2],
    [0, 0, 0, 0, 5, 3, 98, 2],
    [0, 0, 0, 0, 0, 11, 85, 10],
    [2, 3, 0, 22, 0, 0, 3, 3],
    [6, 0, 0, 8, 0, 0, 2, 10],
    [10, 21, 0, 3, 4, 11, 0, 8],
    [8, 10, 3, 1, 55, 3, 11, 0]
]

origins = ("A", "C", "D", "G")
destinations = ("C", "E", "F", "B")

assert_dijkstra(dijkstra, vertices, origins, destinations)
```

```
Data:
         [0, 3, 0, 5, 10, 9, 2, 40]
         [1, 0, 10, 3, 2, 1, 29, 2]
         [0, 0, 0, 0, 5, 3, 98, 2]
         [0, 0, 0, 0, 0, 11, 85, 10]
         [2, 3, 0, 22, 0, 0, 3, 3]
```

```
[6, 0, 0, 8, 0, 0, 2, 10]
[10, 21, 0, 3, 4, 11, 0, 8]
[8, 10, 3, 1, 55, 3, 11, 0]
```

```
Origine: A, Destination: C
"Bonne réponse!"
Origine: A, Destination: E
"Bonne réponse!"
Origine: A, Destination: F
"Bonne réponse!"
Origine: A, Destination: B
"Bonne réponse!"
Origine: C, Destination: C
"Bonne réponse!"
Origine: C, Destination: E
"Bonne réponse!"
Origine: C, Destination: F
"Bonne réponse!"
Origine: C, Destination: B
"Bonne réponse!"
Origine: D, Destination: C
"Bonne réponse!"
Origine: D, Destination: E
"Bonne réponse!"
Origine: D, Destination: F
"Bonne réponse!"
Origine: D, Destination: B
"Bonne réponse!"
Origine: G, Destination: C
"Bonne réponse!"
Origine: G, Destination: E
"Bonne réponse!"
Origine: G, Destination: F
"Bonne réponse!"
Origine: G, Destination: B
"Bonne réponse!"
```

[ ]:

[ ]:

Exercice 2

November 1, 2019

1 Exercice 2:

1.1 Arbres

Un `arbre` est un graphe dans lequel les relations ne peuvent aller que dans un sens (parent - enfant.) La semaine dernière, nous avons vu les arbres binaires, cette semaine nous allons voir les arbres à un sens plus large.

1.2 Utilité

Les arbres ont beaucoup d'utilités: - Ils permettent de maintenir des hiérarchies (la structure des dossiers/fichiers dans votre ordinateur est un arbre.) - Ils permettent de créer des arbres de décision qui permettent d'évaluer un choix et quels conséquences celui-ci va engendrer. - Les interfaces utilisateurs sont pour la plupart basés sur des arbres. - etc.

1.3 Créer un arbre

Pour créer un arbre, utilisez le constructeur `Arbre()`, pour récupérer la `racine` de l'arbre, utilisez `arbre.root`. Pour ajouter un enfant à un noeud, utilisez `node.add_child(child)`, et pour créer un nouveau noeud, utilisez le constructeur `Node(value)`.

```
arbre = Arbre()
racine = arbre.root
child = Node("first child")
racine.add_child(child)
```

[1]:
```
from Arbre import Arbre, Node

arbre = Arbre()
racine = arbre.root
child = Node("first child")
racine.add_child(child)
print(racine)
```

```
{value: root, children: [{value: first child, children: []}]}
```

1.4 Exercice:

En statistiques, on crée souvent des arbres de probabilité pour déterminer la probabilité qu'un évènement arrive `N` fois d'affilée.

1.4.1 Recréez l'arbre de ce diagramme en implémentant votre propre algorithme avec une profondeur de 10 éléments (`"pile"` et `"face"` en minuscules)

[2]:
```
from Arbre import Node

# VOTRE CODE ICI
def add_children(node, max_depth, depth = 0):
    if depth == max_depth:
        return
    node_pile = Node("pile")
    node.add_child(node_pile)
    node_face = Node("face")
    node.add_child(node_face)
    add_children(node_pile, max_depth, depth + 1)
    add_children(node_face, max_depth, depth + 1)
```

[3]:
```
from assertion import assert_arbre_prob
from Arbre import Arbre

arbre_prob = Arbre()
root = arbre_prob.root

add_children(root, 10)

assert_arbre_prob(arbre_prob)
```

```
"Bonne réponse!"
```

1.5 Exercice:

Ecrivez un algorithme qui trouve si une valeur est dans un arbre et retournez le `noeud` qui lui correspond.

Indice: Utilisez l'algorithme `depth-first search` vu en classe.

[6]:
```
# VOTRE CODE ICI
def df_search(node, value):
    if node.value == value:
        return node
    for child in node.children:
        child = df_search(child, value)
        if child is not None:
            return child
    return None
```

[7]:
```
from Arbre import Arbre, Node

arbre = Arbre("Jean")
```

```
jean = arbre.root

marc = Node("Marc")
erica = Node("Erica")
denise = Node("Denise")
henry = Node("Henry") #
melanie = Node("Melanie")
leo = Node("Leo")
stephane = Node("Stephane")
laura = Node("Laura")
josephine = Node("Josephine")

laura.add_child(josephine)
stephane.add_child(laura)
melanie.add_child(stephane)
melanie.add_child(leo)
jean.add_child(marc)
jean.add_child(erica)
erica.add_child(denise)
jean.add_child(henry)
marc.add_child(melanie)

print("Recherche de Laura:")
print(df_search(arbre.root, "Laura"))
print("Recherche de Benoit")
print(df_search(arbre.root, "Benoit"))
```

```
Recherche de Laura:
{value: Laura, children: [{value: Josephine, children: []}]}
Recherche de Benoit
None
```

1.6 Exercice

Écrivez une fonction qui trouve la probabilité d'effectuer une série donnée en utilisant votre arbre (`("pile", "pile", "face", "pile")` par exemple.)

[34]:
```
def probability(series, node):
    def recursive(series, current_series, node):
        if node is None:
            # Fin de l'arbre
            return 0, 0
        count = 0
        explored_pathes = 0
        if len(series) == len(current_series):
            # Si la taille notre série actuelle est la même que celle que l'on␣
 ↪recherche, on la copie et on retire
            # 1 élément
```

```
            next_series = [current_series[i] for i in range(1,␣
↪len(current_series))]
            explored_pathes += 1
            # Le nombre de chemins de même taille augmente de 1
            if series == current_series:
                # Si notre série est la même que celle que l'on cherchait,␣
↪incrémente notre résultat de 1
                count += 1
        else:
            # Si la taille était inférieure, on copie toute la liste et on␣
↪n'incrémente rien
            next_series = [*current_series]
        for child in node.children:
            new_series = [*next_series, child.value]
            # On rappelle récursivement la fonction avec tous les enfants du␣
↪noeud actuel
            pathes_found, pathes_explored = recursive(series, new_series, child)
            # On rajoute les valeurs de notre récursion à nos compteurs actuels
            count += pathes_found
            explored_pathes += pathes_explored
        return count, explored_pathes

    found, total = recursive(series, [], node)
    # return le nombre d'observations trouvées divisé par le total
    return found/total
```

[43]:
```
series = (
    [],
    ["pile"],
    ["pile", "face"],
    ["face", "pile"],
    ["face", "face"],
    ["pile", "pile", "face", "pile"],
    ["pile", "pile", "face", "face"],
    ["face", "face", "face", "face", "face", "face"],
    ["face", "face", "face", "face", "face", "face", "face", "face", "face",␣
↪"face"]
)

for t in series:
    print(probability(t, arbre_prob.root))
```

```
1.0
0.5
0.25
0.25
```

```
0.25
0.0625
0.0625
0.015625
0.0009765625
```

[ ]:

Minimum Spanning Tree

November 1, 2019

0.1 Answers: Minimum Spanning Tree

[ ]:
```
# Partie B
t = [Girdwood, Homer, Talkeenta, Sitka, Gustavus, Skagway]
# Reponse illustrée avec code en partie C
```

0.2 Answers: Kruksal's algorithm

[ ]:
```
# Partie A
num_liens = 5
liens = [('A','C'), ('A', 'F'), ('A', 'D'), ('B', 'F'), ('E','F')
poids_MST = 20 #somme des poids des liens selectionnés
num_steps = 9
list_steps = [('A','D'), ('A', 'C'), ('B', 'F'), ('E', 'F'), ('C','D'), ('A',␣
 ↪'F'), ('A', 'B'), ('D', 'F'), ('C', 'E')
# C-D, A-B, D-F et C-E pas pris car sinon cycle formé
# Réponse illustrée sur image B
```

0.3 Answers: Social Network analysis

[ ]:
```
# Partie A
edge = 'lien amitié'
vertice = 'personne'
dirige = 'NON'
r1 = 'Celui qui a le degrès le plus important'
r2 = 'Lien entre ces amis'
r3 = 'pas de lien entre ami de mon ami et moi'
r4 = 'degrès 0'
```

[ ]:
```
# Partie B
solutions = [ '3C', '5E', '2B', '1A']
```