DOMINGUEZ Ugo
PERDIEUS Antony

RAPPORT
SAE Graphe :
A la conquête d'Hollywood

2023-2024

SOMMAIRE

Introduction:

Dans le contexte d'une SAE afin d'appliquer nos connaissances et de les améliorer, nous avions pour objectif d'implémenter un programme permettant de répondre à un certain nombre de requêtes effectuées sur des jeux de données. Ce programme reprend l'idée du site The Oracle Of Bacon qui consiste à calculer le nombre de Bacon d'un acteur. Le nombre de Bacon d'un acteur correspond à la distance (nombre d'arêtes) entre le sommet représentant l'acteur et le sommet représentant Kevin Bacon dans un graphe. Les différentes fonctions à implémenter ont donc comme but d'obtenir la centralité d'un acteur, et son nombre de Bacon. Il est également demandé de tester différentes implémentations de nos fonctions, et de choisir celle qui est la plus efficace.

Tâches et organisation:

Tout d'abord, nous avons mis en place un GitHub, nous permettant de mettre notre code en commun. Suite à celà, nous nous sommes réparties les tâches principales, telles que la réalisation des tests, l'implémentation des requêtes, ou celle du menu de consultation. Dans un premier temps, Ugo s'occupait des tests et de la documentation, pendant qu'Antony se préoccupait de la compréhension des implémentations et de la fonction *json_vers_nx*. Après avoir réalisé ces travaux à faire en amont du code pur, nous avons décidé des fonctions que chacun aimerait réaliser, celles pour lesquelles nous avions une idée préalable. Voici donc un tableau des fonctions réalisés par chacun:

Ugo	Antony
`collaborateurs_proches`	`json_vers_nx`
`distance`	`collaborateurs_communs`
`centralite`	`est_proche`
+ test_requetes.py	`distance_naive`
+ const.py	`centre_hollywood`
+ oracle.py	`eloignement_max`

Après avoir réalisé différentes implémentations de chacune des requêtes, nous nous sommes enfin penché sur le menu de consultation (oracle.py). Ugo a donc réalisé la majeure partie du code de ce fichier, mais Antony est tout de même venu le compléter avec de nouvelles idées.

Réponses:

6.2:

En termes de théorie des graphes, l'ensemble des collaborateurs en commun de deux acteurs/actrices donné.e.s, peut être exprimé comme l'intersection des ensembles de voisins de ces deux sommets.

La complexité de cette fonction dépend du degré du sommet u, en plus du degré sur sommet v, donc O(deg(u) + deg(v)).

6.3:

L'algorithme classique en théorie des graphes qui est au cœur de ce programme est un parcours en largeur (BFS).

Pour déterminer si un acteur u se trouve à distance k d'un autre acteur v, il suffit de vérifier si l'acteur v, se trouve dans les collaborateurs proches de l'acteur u à la distance k.

Ré-utiliser la fonction *est_proche*, me paraît être une mauvaise idée, car il pourrait y avoir trop de parcours en largeur exécutés à la suite. La complexité d'un tel algorithme est de O(n*m), avec n égal au nombre de sommets, et m égal au nombre d'arêtes du graphe.

La complexité de la fonction *distance* est de O(n+m), avec n égal au nombre de sommets, et m égal au nombre d'arêtes du graphe.

6.4:

La notion de théorie des graphes qui intervient dans la fonction *centralite* s'appelle l'excentricité. L'excentricité d'un sommet u est définie comme étant la distance la plus longue entre ce sommet et tous les autres sommets du graphe.

6.5:

Performances:

Pour ce qui est des performances de chacune des fonctions, nous nous sommes concentré sur les principales: *distance* et *centralite*. Tout de même, nous avons essayé d'optimiser le plus possible les fonctions que nous avons implémenter. Nous nous sommes penché sur une version de *centralite*, en utilisant la fonction *distance*. Cependant, malgré tous nos efforts pour améliorer la fonction distance, la fonction *centralite* n'a jamais été assez optimisée pour obtenir un résultat à l'appel des fonctions *centre_hollywood* et *eloignement_max* (utilisant *centralite*). Voici quelques implémentations de la fonction *distance*:

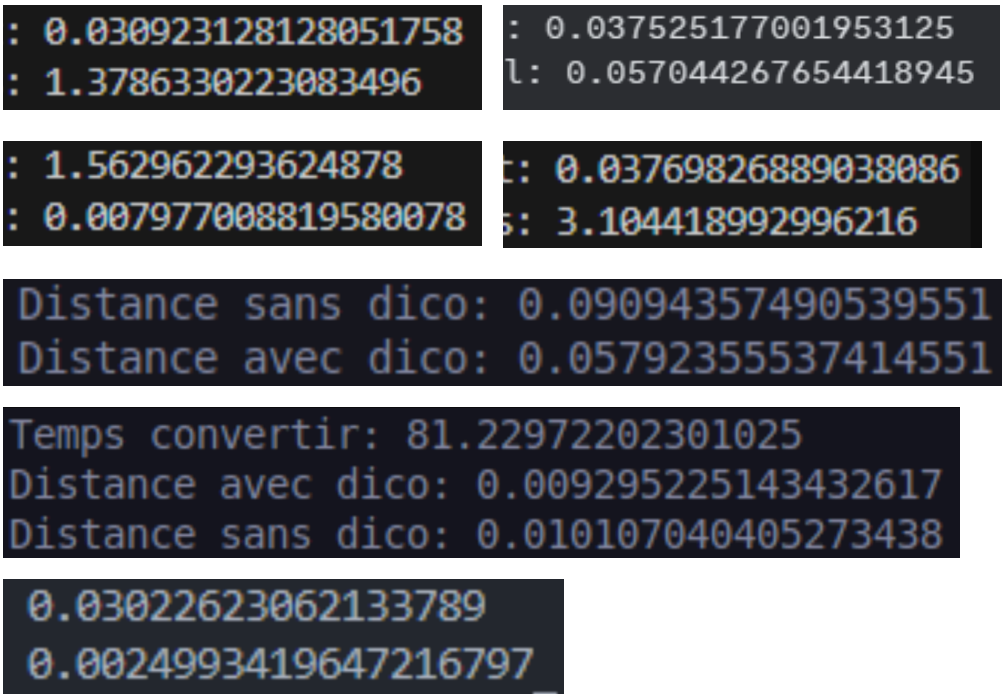
```python
def distance(G, u, v):
    if u not in G or v not in G:
        return -1

    if u == v:
        return 0

    a_parcourir = [u]
    distances = {u: 0}

    while a_parcourir:
        courrant = a_parcourir.pop(0)

        for voisin in G[courrant]:
            if voisin not in distances:
                distances[voisin] = distances[courrant] + 1

                if voisin == v:
                    return distances[voisin]

                a_parcourir.append(voisin)
```

```python
def distance(G: nx.Graph, u: str, v: str) -> int:
    if u not in G.nodes or v not in G.nodes:
        return -1

    if u == v:
        return 0

    collaborateurs = {u}
    a_traiter = {u}

    for d in range(nx.number_of_nodes(G)):
        seront_traites = set()

        for sommet in a_traiter:
            voisins = set(G.adj[sommet])
            seront_traites |= voisins - collaborateurs

        collaborateurs |= seront_traites
        a_traiter = seront_traites

        if v in collaborateurs:
            return d + 1

    return -1
```

```python
def distance(G, u, v):
    if u not in G or v not in G:
        return -1

    if u == v:
        return 0

    niveau = 0
    niveau_actuel = {u}
    niveau_suivant = set()

    while niveau_actuel:
        courant = niveau_actuel.pop()

        for voisin in G[courant]:
            if voisin not in niveau_actuel:
                if voisin == v:
                    return niveau + 1

                niveau_suivant.add(voisin)

        if not niveau_actuel:
            niveau += 1
            niveau_actuel = niveau_suivant
            niveau_suivant = set()

    return -1
```

```python
def distance(G: nx.Graph, u: str, v: str) -> int:
    if u not in G.nodes or v not in G.nodes:
        return -1

    if u == v:
        return 0

    a_traiter = {u}
    collaborateurs = {u}

    d = 0
    while a_traiter:
        seront_traites = set()

        for sommet in a_traiter:
            if v == sommet:
                return d

            voisins = set(G.adj[sommet])
            seront_traites |= voisins - collaborateurs

        collaborateurs |= seront_traites
        a_traiter = seront_traites

        d += 1

    return -1
```

Dans le but de tester l'efficacité de ces différentes implémentations, nous avons simplement comparé le temps d'exécution de ces dernières en utilisant le module python time. Pour celà, nous avons utilisé le jeu de données contenant 140 000 films, et choisi deux acteurs aléatoires. Après quelques essais, nous avons gardé la version qui avait le temps d'exécution le plus court.



Malgré ces nombreuses modifications, la fonction *distance* n'est pas assez efficace pour être utilisée dans la fonction *centralite*. Nous avons également pensé à une implémentation utilisant l'algorithme de Djikstra. Une autre idée m'est aussi venu en tête: Parcourir parallèlement le graphe, en partant d'un sommet u et d'un autre sommet v. Ces deux parcours devront obligatoirement de rejoindre au bout d'un moment, ce qui donnera la distance entre les deux sommets. Nous avons essayé de réaliser ces solutions, mais aucune de nos implémentations n'a été concluante.

Conclusion:

Cette SAE graphe s'est bien déroulée de notre côté, grâce à une organisation d'équipe efficace. Nous avons effectué la majorité des tâches demandés, chacune des fonctions à faire, les tests associés, et un menu de consultation pour interagir avec les méthodes principales du programme. Malgré la complétion de ces tâches, et les essais d'optimisation de certaines fonctions, quelques-unes d'entre elles manquent d'efficacité sur des jeux de données trop grands.