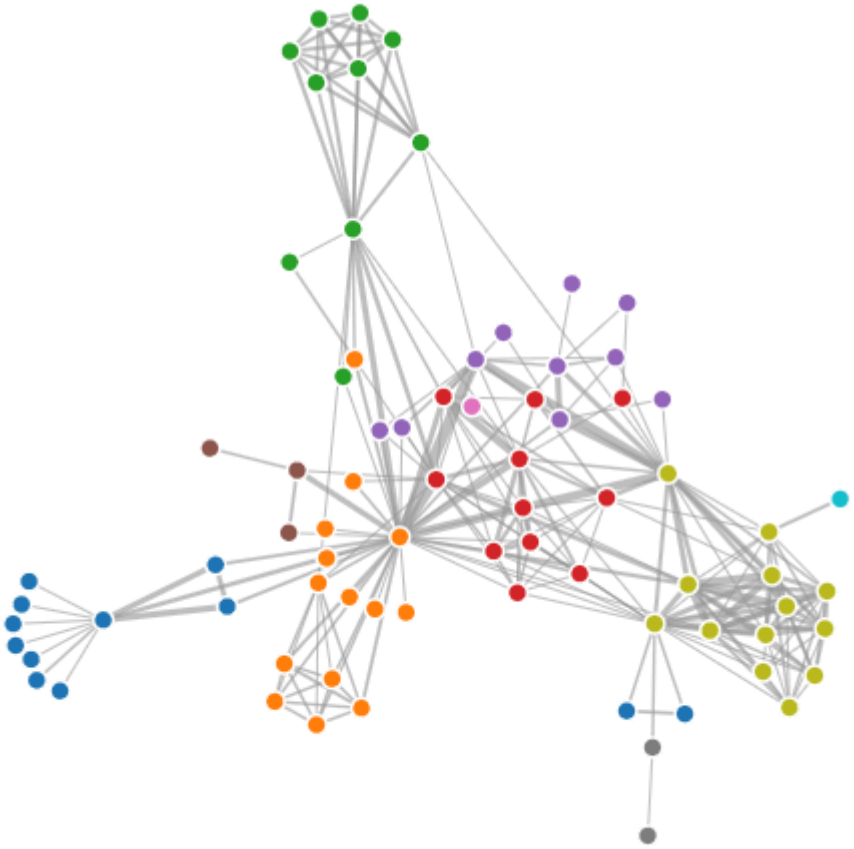
Niksan Nagarajah
Alexy Wiciak
1-1A

UNIVERSITE D'ORLEANS

IUT ORLÉANS

Rapport
SAE Graphes



Année 2023-2024

Table des matières

I) Travail réalisé :

Au début de cette SAÉ, nous nous sommes concentrés sur la compréhension du sujet. Ensuite, nous avons cherché à réaliser les algorithmes demandés en les partageant sur git.

Nous avons fait les fonctions de manière indépendante afin de se concentrer sur des tâches précises. Nous avons aussi dû faire des recherches liées à NetworkX afin d'implémenter son vocabulaire et ceci nous a permis de réaliser les fonctions demandées plus facilement.

Par la suite, nous avons fait un fichier test, afin de s'assurer de la fiabilité du code produit. Ensuite, nous avons fait une interface graphique dans le terminal qui permet de demander à l'utilisateur les résultats des différentes fonctions à travers des menus.

Enfin, nous avons rédigé notre rapport individuel et collectif en répondant aux questions suivantes, et en détaillant les points d'amélioration.

II) Réponses liées au sujet :

6.2) Comment exprimez-vous cette notion (ensemble des collaborateurs en commun) en termes de théorie des graphes?

Dans le graphe, chaque collaborateur est représenté par un sommet.
Une arête (ligne) est tracée entre deux sommets si les deux collaborateurs ont travaillé ensemble sur au moins un film.
En termes de théorie des graphes, cette notion (ensemble des collaborateurs en communs) correspond à l'intersection des voisins du premier acteur et les voisins du deuxième acteur dans le graphe.

Voici l'extrait de notre code :

```python
def collaborateurs_communs(G, u, v):
    """
    Retourne l'ensemble des acteurs qui ont collaboré avec à la fois acteur1 et acteur2
    dans les films répertoriés dans le fichier JSON G.

    Paramètres :
    - G : (Graph) Graphe considéré.
    - v : (str) Nom de l'acteur 2.

    Returns :
    - (set) Ensemble des acteurs qui ont collaboré avec acteur1 et acteur2.
    """
    #Complexité : O(N)
    # start = time.time()
    try :
        collaborateurs = set()
        colab_acteur1 = set(G[u].keys())
        colab_acteur2 = set(G[v].keys())
        for acteur in colab_acteur1: #O(N)
            if acteur in colab_acteur2 and acteur != u and acteur != v:
                collaborateurs.add(acteur)
        # print(time.time()-start)
        return collaborateurs
    except:
        print("Un des noms des acteurs spécifié est incorrecte ou n'est pas dans notre base de données\n")
        # print(time.time()-start)
        return None
```

Pouvez-vous donner une borne inférieure sur le temps nécessaire à l'exécution de votre fonction?

Pour le fichier data.json, notre algorithme met 0.0001881122589111328 secondes à s'exécuter, et pour data_100.json, elle met 1.1920928955078125e-05 secondes.

6.3) Étudiez ce code.

Pour chaque voisin de l'acteur u, la fonction ajoute les voisins de ces derniers. Cet algorithme explore les nœuds du graphe niveau par niveau à partir du nœud source, soit l'acteur u.

Voici l'extrait du code :

```python
# Code donné
def collaborateurs_proches(G,u,k):
    """Fonction renvoyant l'ensemble des acteurs à distance au plus k de l'acteur u dans
    le graphe G. La fonction renvoie None si u est absent du graphe.

    Parametres:
        G: le graphe
        u: le sommet de départ
        k: la distance depuis u
    """
    #Complexité : O(N)³
    if u not in G.nodes:
        print(u,"est un illustre inconnu")
        return None
    collaborateurs = set()
    collaborateurs.add(u)
    # print(collaborateurs)
    for _ in range(k): # O(N)
        collaborateurs_directs = set()
        for c in collaborateurs: # O(N)
            for voisin in G.adj[c]: # O(N)
                if voisin not in collaborateurs and voisin != u:
                    collaborateurs_directs.add(voisin)
        collaborateurs = collaborateurs.union(collaborateurs_directs)
    return collaborateurs
```

Reconnaissez-vous l'algorithme classique en théorie des graphes qui est au cœur de ce programme?

L'algorithme classique en théories des graphes qui est au cœur de la fonction collaborateurs_proches est l'algorithme de recherche en largeur. En effet, à chaque tour de boucle de la fonction collaborateurs_proches, on explore une nouvelle couche de voisins.

Grâce à la fonction précédente, comment pouvez-vous déterminer si un acteur se trouve à distance k d'un autre acteur?

Pour cela, à l'aide de la fonction précédente, on récupère l'ensemble des voisins qui sont proches de l'acteur à un degré k. Ensuite, on regarde si l'autre acteur est dans cet ensemble. Si c'est le cas, alors on peut dire que l'autre acteur est à une distance au plus k de l'acteur initial. Sinon, l'autre acteur n'est pas à une distance au plus k de l'acteur initial.

Voici l'extrait de notre code :

```python
def est_proche(G,u,v,k=1):
    """
    La fonction permet de savoir si le collaborateur 'u' est proche de 'v'.

    Paramètres :

    - G : (Graph) Graphe considéré.
    - u : (str) Nom de l'acteur 1.
    - v : (str) Nom de l'acteur 2.
    - k : (int) distance proche

    Returns:
        (bool): True si la distance entre les 2 acteurs est considéré comme proche.
    """
    #Complexité : O(N)³
    lesColaborateurs = collaborateurs_proches(G, u, k) #O(N)³ : appel de la fonction précédente
    if lesColaborateurs != None:
        if v in lesColaborateurs:
            return True
    return False
```

Tentons maintenant de déterminer la distance entre deux acteurs. Est-ce que réutiliser la fonction précédente vous semble intéressant? Donnez la complexité (asymptotique) d'un tel algorithme.

Pour cela, on peut utiliser la fonction précédente mais elle peut prendre beaucoup de temps. Pour cela, il faudrait appeler la fonction précédente pour récupérer l'ensemble des voisins à distance k et vérifier si l'autre acteur est dans cette ensemble. Et si ce n'est pas le cas, il faudrait augmenter la distance de 1 à chaque fois qu'il ne fait pas partie de l'ensemble jusqu'à le trouver.
La complexité d'un tel algorithme serait de $O(N)^4$.

Voici l'extrait de notre code :

```python
def distance_naive(G, u, v):
    #Complexité : #O(N)⁴
    if u not in G.nodes() or v not in G.nodes():
        return None
    degre = 0
    while not est_proche(G, u, v, degre): #O(N)⁴ # PAS MéTHODE ADAPTE (privilégié dico)
        degre += 1
    return degre
```

Comment pouvez-vous maintenant modifier la fonction qui vous a été fournie afin de trouver la distance entre deux acteurs? Donnez la complexité d'un tel algorithme.

Pour cela, il faudrait pouvoir stocker dans un dictionnaire : à telle distance, on a tels voisins ({0 : acteur}, 1 : {l'ensemble des voisins de acteur}, ...). Ainsi, pour chaque distance, on aurait les voisins des voisins. Et si à un moment, on trouve parmi les voisins l'autre acteur, alors il faudrait retourner à quelle distance il se trouve.
La complexité d'un tel algorithme serait de $O(N)^3$.

Voici l'extrait de notre code :

```python
def distance(G, u, v):
    """
    La fonction permet trouver la distance entre 2 acteurs.

    Paramètres :

    - G : (Graph) Graphe considéré.
    - u : (str) Nom de l'acteur 1.
    - v : (str) Nom de l'acteur 2.

    Returns:
        (int): La distance entre les 2 acteurs.
    """
    #Complexité : #O(N)³
    if u == v:
        return 0
    ensemble_colab = set()
    dico_collab = {0 : {u}}
    i = 0
    while dico_collab[i] != set(): #O(N)
        dico_collab[i+1] = set()
        for acteur in dico_collab[i]: #O(N)
            colab = G.edges(acteur)
            for (comedien1, comedien2) in colab: #O(N)
                if comedien1 not in ensemble_colab:
                    ensemble_colab.add(comedien1)
                    dico_collab[i+1].add(comedien1)
                if comedien2 not in ensemble_colab:
                    ensemble_colab.add(comedien2)
                    dico_collab[i+1].add(comedien2)
            if v in dico_collab[i+1]:
                return i+1
            # print(colab)
        i += 1
    return None
```

6.4) Quelle notion de théorie des graphes permet de modéliser cela ?

L'excentricité d'un sommet v dans un graphe est définie comme étant la plus grande distance entre v et n'importe quel autre sommet du graphe.
Ainsi, c'est l'excentricité qui est mise en avant ici. En réalité, un sommet avec une faible excentricité est considéré comme étant plus au centre, c'est-à-dire qu'il est relativement proche de tous les autres sommets, alors qu'un sommet avec une grande excentricité est plus périphérique.

Voici nos deux extraits de code :

```python
def centralite(G, u):
    """
    Déterminer la plus grande distance qui le sépare d'un autre acteur dans le graphe.

    Args:
        G (nx.Graph): Le graphe des acteurs.
        u : (str) Nom de l'acteur

    Returns:
        int : Distance maximale qui sépare l'acteur u d'un autre acteur dans le graphe.
    """
    # Complexité : #O(N)²
    if u not in G.nodes():
        return None
    # # return G.degree(u)
    # # ou
    # lengths = nx.shortest_path_length(G, source=u)
    # return max(lengths.values())
    distances = {u : 0} # Parcours en largeur
    a_faire = [u]
    while a_faire: #O(N)
        courant = a_faire.pop(0)
        distance_actuel = distances[courant]
        for voisin in G.neighbors(courant): #O(N)
            if voisin not in distances.keys():
                distances[voisin] = distance_actuel + 1
                a_faire.append(voisin)
    return max(distances.values())
```

```python
def centre_hollywood(G):
    """
    Déterminer l'acteur le plus central du graphe.

    Args:
        G (nx.Graph): Le graphe des acteurs.

    Returns:
        str : Le nom de l'acteur au centre d'Hollywood
    """
    # acteur_maxi = None
    # maxi = 0
    # for acteur in G.nodes():
    #     if centralite(G, acteur) > maxi:
    #         acteur_maxi = acteur
    #         maxi = centralite(G, acteur)
    # return acteur_maxi, maxi
    #Complexité : O(N)³
    start = time.time()
    les_centralites = {}
    for acteur in G.nodes(): #O(N)
        les_centralites[acteur] = centralite(G, acteur) #O(N)²
    acteur_central = None
    mini = None
    for (acteur, centralite_acteur) in les_centralites.items(): #O(N)
        if mini == None or centralite_acteur < mini:
            mini = centralite_acteur
            acteur_central = acteur
    print(time.time() - start)
    return acteur_central
```

6.5) Est-ce que ce nombre est bien inférieur ou égal à 6 pour le jeu de données fourni?

Il nous est difficile voir impossible de pouvoir tester la fonction eloignement_max avec le jeu de données fourni car chaque tour de boucle nous prend environ 14-15 secondes.

```python
def eloignement_max(G:nx.Graph):
    """…
    #Complexité : O(N)³
    print(len(G.nodes()))      You, now • Uncommitted changes
    les_centralites = {}
    for acteur in G.nodes(): #O(N)
        d = time.time()
        les_centralites[acteur] = centralite(G, acteur) #O(N)²
        print("Centralité de", acteur, "en", time.time() - d)
    return max(les_centralites.values())

# print(eloignement_max(G))
print(eloignement_max(json_vers_nx("./donnees/data.json"))) # Trop long
```

```
o22301527@info23-09:~/Documents/SAE/SAE_Exploration_Algori
e/iut45/Etudiants/o22301527/Documents/SAE/SAE_Exploration_
429254
Centralité de Núria Espert en 13.516013145446777
Centralité de Rosa Maria Sardà en 13.79999303817749
Centralité de Anna Lizaran en 13.952106952667236
Centralité de Mercè Pons en 13.98882794380188
Centralité de Bruce Campbell en 13.629961729049683
Centralité de Embeth Davidtz en 13.714652061462402
Centralité de Marcus Gilbert en 13.613785743713379
Centralité de Ian Abercrombie en 13.660487651824951
Centralité de Richard Grove en 13.476224422454834
Centralité de Timothy Patrick Quill en 13.691880941390991
Centralité de Michael Earl Reid en 13.802758693695068
```

Sachant qu'il y a 429254 noeud dans le graphe, le temps d'exécution de la fonction prend : 15*429254 = 6438810 secondes soit : 6438810 / 60 = 107313.5 min soit : 1073131.5 / 60 = 1788.5583 heure soit 1788.5583 / 24 = 74.523 jours.
De ce fait, sans avoir optimisé le code, il nous est impossible de répondre à cette question avec le jeu de données fourni.
De même, cela prend également beaucoup de temps pour la data 10000.

Néanmoins, nous avons testé avec les données 100 et 1000.

```python
def eloignement_max(G:nx.Graph):
    """…
    #Complexité : O(N)³
    les_centralites = {}
    for acteur in G.nodes(): #O(N)
        d = time.time()
        les_centralites[acteur] = centralite(G, acteur) #O(N)²
        # print("Centralité de", acteur, "en", time.time() - d)
    return max(les_centralites.values())

print("Pour data_100, l'éloignement max est de", eloignement_max(json_vers_nx("./donnees/data_100.json")))
print("Pour data_1000, l'éloignement max est de", eloignement_max(json_vers_nx("./donnees/data_1000.json"
```

On obtient :

```
o22301527@info23-09:~/Documents/SAE/SAE_Expl
527/Documents/SAE/SAE_Exploration_Algorithmi
Pour data_100, l'éloignement max est de 3
Pour data_1000, l'éloignement max est de 4
```

Voici les complexités des fonctions réalisées :

txt_to_json : $O(N)^3$
json_vers_nx : $O(N)^3$
collaborateurs_communs : $O(N)$
collaborateurs_proches : $O(N)^3$
est_proche : $O(N)^3$
distance_naive : $O(N)^4$
distance : $O(N)^3$
centralite : $O(N)^2$
centre_hollywood : $O(N)^3$
eloignement_max : $O(N)^3$

III) Évaluation expérimentale :

Mesure de performance :

En moyenne, la performance des fonction réalisé est la suivante :

```
Pour la fonction collaborateur_communs 0.00019288063049316406
Pour la fonction collaborateurs_proches 1.3153955936431885
Pour la fonction collaborateurs_proches 0.0001513957977294922
Pour la fonction collaborateurs_proches 1.9073486328125e-06
Pour la fonction est_proche 4.458427429199219e-05
Pour la fonction collaborateurs_proches 0.00011682510375976562
Pour la fonction est_proche 0.00017881393432617188
Pour la fonction collaborateurs_proches 0.05069231986999512
Pour la fonction distance_naive 0.0534081459045410 16
Pour la fonction distance 0.015808820724487305
(9, ('Pour la foccntion centralite ', 18.956682205200195))
```

Cet affichage est obtenu suite à l'exécution de toutes les fonctions avec le jeu de données fourni (data.txt). De plus, toutes les valeurs affichées ci-dessus sont exprimées en seconde. Néanmoins, les deux dernières fonctions (centre_holywood et eloignement_max) n'ont pu être affichées dû au temps qu'elle prenait pour être exécutées.

Analyse :

Selon nous, nous n'avons pas réussi à optimiser toutes les fonctions. C'est pourquoi nous estimons que nos fonctions prennent plus de temps que prévu pour donner le résultat attendu.
Néanmoins, nous avons tout de même réalisé plusieurs versions des fonctions pour garder les fonctions les plus performantes. Certaines des autres fonctions, solutions sont en commentaire dans le fichier requêtes.py.

IV) Solutions et points d’amélioration:

Avec plus de temps, nous aurions pu améliorer l’efficacité de nos programmes afin de mettre moins de temps à chaque exécution de fonctions. Cela aurait permis de réduire la complexité des fonctions concernées.

De plus, nous aurions pu ajouter plus de tests pour traiter les exceptions afin de garantir une fonctionnalité optimale sur l’ensemble du projet.

Enfin, nous aurions pu améliorer notre efficacité lié à git en résolvant certains conflits plus rapidement, ce qui nous a pris beaucoup de temps sur la SAE en générale. C’est d’ailleurs ce temps qui nous a manqué pour optimiser nos fonctions.

V) Conclusion

En conclusion, cette SAÉ nous a permis de nous améliorer sur différents domaines. Tout d’abord, nous nous sommes améliorés sur le travail d’équipe en coopérant sur github et en cherchant les réponses ensembles.

Ensuite, nous avons pu approfondir nos connaissances en programmation Python, en travaillant sur des concepts avancés tels que NetworkX lié aux graphes par exemple.

Nous avons également renforcé notre aptitude à résoudre des problèmes complexes et à assimiler des spécifications détaillées, en ajustant nos solutions aux exigences spécifiques du projet.

Enfin, cette SAÉ a été une expérience enrichissante qui nous a non seulement permis de développer des fonctionnalités, mais aussi de consolider un large éventail de compétences, fondamentales pour notre futur et notre évolution professionnelle.