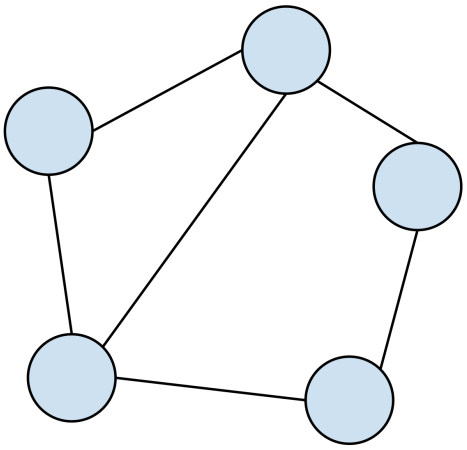
THUAULT Corentin
MEUNIER Axel

SAE 2.02
Exploration Algorithmique d'un Problème

The Oracle Of Bacon



Reformulation :

Pour cette SAE 2.02, nous devions réaliser une “exploration algorithmique d’un problème”..

Le sujet de l'exercice était le suivant : un jeune passionné de cinéma et de sport ambitionnait de devenir un coach sportif renommé à Hollywood en se faisant embaucher par des acteurs ou actrices influents, espérant ainsi augmenter sa notoriété grâce à leur exposition médiatique. Pour accélérer le processus, il souhaitait identifier les candidats les plus propices à sa réussite. Pour ce faire, il s'est inspiré du concept des "six degrés de Kevin Bacon", une théorie des graphes popularisée dans les années 90, où chaque acteur est lié à Kevin Bacon par un certain nombre de collaborations cinématographiques.

En s'appuyant sur cette idée, nous avons été chargés de mettre en place un programme similaire à "The Oracle Of Bacon" permettant d'évaluer la centralité des acteurs d'Hollywood en Python, en utilisant la bibliothèque Networkx pour représenter le graphe des collaborations.

Travail réalisé :

Durant ces deux derniers mois nous avons réalisé les différentes fonctions qu’il nous était demandé d’implémenter et nous les avons optimisées au maximum que nos connaissances pouvaient nous le permettre. Nous avons réussi à faire toutes les fonctions qui nous étaient demandées et même à les faire évoluer au fur et à mesure que notre compréhension du sujet augmentait. Nous avons fait différentes versions pour chaque fonctions permettant de régler les problèmes des précédentes et nous avons terminé en faisant des fonctions optimisées utilisant la bibliothèque de networkX. Nous avons implémenté ces fonctions à la fin de notre fichier “requêtes” car nous considérons ces fonctions comme des bonus plutôt que comme de réelles réponses aux questions qui nous ont été posées, après tout nous n’avons pas fait les fonctions se trouvant dans la bibliothèque networkX.

Nous avons, pour chacune des fonctions, réalisé des tests avec un Graphe plus simple à comprendre que les données qui nous ont été fournies avec le sujet. En addition avec ces tests nous avons réalisé un IHM sur terminal permettant à l’utilisateur de choisir les données voulues puis différentes options tel que le calcul de la distance entre les deux acteurs entrés en paramètres ou encore la centralité du graphe choisi (parmis les différents fichiers de données, 100, 1000, 10000 et le complet). Nous avons ajouté à notre rendu un fichier “Versions.py” qui regroupe différentes versions de différentes fonctions que nous avons réalisées.

Partage des tâches :

Passons maintenant au partage du travail. Nous avons tous les deux participé à la création des fonctions et leur optimisation. Le tableau ci-dessous permet de présenter de manière claire la manière dont nous avons partagé le travail.

	Code	Optimisation	Tests	IHM	Rapport
Corentin	X	X			X
Axel	X	X	X	X	

Fonctions réalisées :

Corentin : collaborateurs_communs_Bonus; est_proche; distance_naive; distance; eloignement_max, centralite_groupe

Axel : collaborateurs_communs; centralite; centre_hollywood;

A deux : json_vers_nx;

Fonctions optimisées :

Corentin : collaborateurs_communs; est_proche; distance; eloignement_max

Axel :

A deux : json_vers_nx; collaborateurs_communs; centralite; centre_hollywood;

Réponses aux questions :

Passons maintenant aux différentes questions qui sont posées sur le sujet. Tout d'abord, pour les complexités, nous les avons placées dans les commentaires de chaque fonction donc nous ne les citerons pas dans ce rapport sauf si elles sont demandées explicitement dans les questions. Nous allons cependant répondre au reste des questions ici :

6.2 : En termes de théorie des graphes, l'ensemble des collaborateurs communs entre deux acteurs peut être exprimé comme l'intersection des ensembles de voisins de ces deux acteurs dans le graphe. Autrement dit, ce sont les acteurs avec lesquels les deux acteurs donnés ont collaboré.

La complexité asymptotique de la fonction “collaborateurs_communs” est donnée par O(d1 + d2), où d1 est le degré de l'acteur u et d2 est le degré de l'acteur v. Cette complexité découle du fait que nous devons obtenir les ensembles de voisins pour chaque acteur (ce qui prend O(d1) et O(d2) opérations respectivement), puis nous devons effectuer une intersection entre ces deux ensembles, ce qui peut être fait en temps linéaire par rapport à la taille de ces ensembles, d'où la complexité totale O(d1 + d2).

Cette complexité est une borne inférieure raisonnable, car il est nécessaire d'examiner au moins tous les voisins de chaque acteur pour trouver les collaborateurs communs, et dans le pire des cas où tous les voisins de u et de v sont différents et l'opération d'intersection serait effectuée sur deux ensembles distincts la recherche de tous les voisins de chaque acteur sera quand-même réalisée.

6.3 : Oui, la fonction "collaborateurs_proches" utilise une variante de l'algorithme de parcours en largeur (BFS) en explorant les voisins d'un sommet jusqu'à une certaine distance k. C'est un algorithme classique en théorie des graphes utilisé pour explorer les voisins d'un sommet dans un graphe.

Pour déterminer si un acteur se trouve à une distance k d'un autre acteur, nous pourrions utiliser la fonction "collaborateurs_communs". En effet, nous pouvons d'abord trouver les acteurs proches des deux acteurs donnés jusqu'à une distance k à l'aide de la fonction "collaborateurs_proches", puis nous chercherons l'intersection de ces ensembles pour trouver les acteurs communs. Si l'acteur que nous voulons vérifier est dans cet ensemble, cela signifie qu'il se trouve à une distance au plus k de l'autre acteur.

Pour déterminer la distance entre deux acteurs, réutiliser la fonction "collaborateurs_proches" pourrait être utile, mais cela dépend du contexte et des besoins spécifiques. Si nous voulons simplement savoir si deux acteurs sont à une distance donnée l'un de l'autre, alors oui, cela pourrait être une approche efficace. Cependant, si nous avons besoin de calculer la distance exacte entre les deux acteurs, une approche plus sophistiquée, comme l'utilisation de l'algorithme de Dijkstra (disponible via la bibliothèque networkX), pourrait être nécessaire.

6.4 : La notion de théorie des graphes qui permet de modéliser cela est la "centralité de l'intermédiarité". Cette mesure de centralité évalue l'importance d'un nœud en fonction du nombre de chemins les plus courts entre tous les autres nœuds qui passent par ce nœud. En d'autres termes, un acteur avec une grande centralité d'intermédiarité est un acteur qui se trouve sur de nombreux chemins entre les autres acteurs dans le graphe. Ainsi, en identifiant le chemin le plus long d'un acteur vers tous les autres acteurs dans le graphe, nous pouvons estimer sa centralité dans le graphe.

6.5 : Nous allons donner l’éloignement max pour chacun des jeux de données :

Jeu 100 : 3

Jeu 1000 : 4

Jeu 10 000 : 4

Jeu Complet : 15

15 > 6, pour le graphe complet l'éloignement max n'est ni égal ni inférieur à 6.

Evaluation Expérimentale :

Dans cette partie je vais présenter les différentes versions de chacune de nos fonctions, expliqué comment nous avons amélioré ces différentes versions et enfin faire une comparaison quant au temps qu'elles mettent à trouver le résultat recherché.
Pour calculer le temps des fonctions nous avons utiliser le module "time" sous cette forme :

```
start = time.time()
print(fonction(G))
end = time.time()
print(end - start)
```

json_vers_nx :

Pour cette fonction nous avons réalisé 3 versions. La première version ne contenait pas la fonction "suppression" permettant de retirer les crochets et les espaces superflus que nous pouvions trouver quand nous prenions la partie "cast" des jeux de données. La deuxième version ne contenait pas d'encodage lors de l'ouverture du fichier, problème que nous avons réglé dans la troisième et dernière version qui contient la ligne "open(chemin, "r", encoding="utf-8")" ce qui permet d'ouvrir correctement le jeu de donnée complet.

collaborateurs_communs :

Pour cette fonction nous avons réalisé 3 versions dont une bonus. Dans la première version de la fonction collaborateurs_communs, nous avons tenté de trouver l'ensemble des acteurs ayant collaboré avec deux acteurs donnés en utilisant la fonction collaborateurs_proches. Cependant, cette approche n'était

pas correcte car nous avons choisi d'utiliser une fonction qui n'était pas encore définie à ce stade du code, ce que nous avons voulu modifier par la suite.

Dans la deuxième version, nous avons corrigé cette erreur en calculant directement l'ensemble des voisins de chaque acteur donné et en prenant l'intersection de ces ensembles pour trouver les acteurs communs. Cela nous permet de déterminer plus efficacement les collaborateurs communs sans avoir besoin de recourir à la fonction collaborateurs_proches.

La troisième version, collaborateurs_communs_Bonus, répond au bonus en renvoyant le sous-graphe induit par l'acteur u et tous les acteurs à une distance au plus k de u dans le graphe G. Contrairement aux deux premières versions qui renvoient un ensemble d'acteurs, cette version renvoie un sous-graphe. Cela peut être utile dans certains cas où nous voulons non seulement connaître les acteurs communs, mais aussi étudier la structure des collaborations dans un sous-ensemble spécifique du graphe.

Temps pour collaborateurs communs entre “Jack Lemmon”, “Charles Durning” sur le jeu de donnée 10 000 :

V1 : 0.00324 secondes

V2 : 0.00132 secondes

est_proche :

Pour cette fonction nous avons réalisé 2 versions. La première version de la fonction est_proche tente de déterminer si l'acteur v se trouve à une distance k de l'acteur u dans le graphe G. Elle utilise une méthode similaire à celle de la fonction collaborateurs_proches, où elle construit un ensemble d'acteurs à une distance k de u en parcourant récursivement les voisins à chaque étape. Cependant, cette approche peut être inefficace car elle répète le même processus pour chaque distance k.

Dans la deuxième version de est_proche, nous avons optimisé l'algorithme en utilisant la fonction collaborateurs_proches définie précédemment. Cette fonction nous permet de trouver efficacement tous les acteurs à une distance k de u dans le graphe G. Ensuite, nous vérifions simplement si l'acteur v se trouve dans cet ensemble d'acteurs proches. Cette approche simplifiée réduit la complexité de la fonction, ce qui la rend plus efficace pour déterminer la proximité entre deux acteurs dans le graphe.

Temps pour est_proche(G, “Jack Lemmon”, “Charles Durning”) sur le jeu de donnée 10 000:

V1: 16 secondes

V2: 14 secondes

distance_naive :

Nous avons réalisé une seule et unique version pour cette fonction qui n’est volontairement pas optimisée pour le calcul de distance. Cette approche naïve consiste à déterminer la distance entre deux acteurs dans un graphe en explorant progressivement les niveaux de voisinage à partir d’un acteur donné.

Pour ce faire, nous utilisons une boucle qui appelle la fonction collaborateurs_proches à plusieurs reprises, en augmentant progressivement la distance jusqu'à ce que l'acteur cible soit trouvé dans l'ensemble des collaborateurs. Nous initialisons la distance à zéro et vérifions si l'acteur cible est un collaborateur direct. Si ce n'est pas le cas, nous augmentons la distance et recherchons à nouveau, jusqu'à ce que l'acteur cible soit trouvé ou que nous ayons parcouru tous les sommets du graphe.

En résumé, cette version met en évidence une approche simple mais inefficace pour calculer la distance entre deux acteurs dans un graphe, en soulignant la nécessité d'optimiser les algorithmes pour des calculs plus rapides et plus efficaces dans des graphes de grande taille.

distance :

Pour cette fonction nous avons réalisé 3 versions. La première version de la fonction distance calcule la distance entre deux acteurs u et v dans le graphe G en utilisant une approche de parcours en largeur (BFS). Elle utilise une file pour explorer les voisins à chaque niveau de la hiérarchie du graphe jusqu'à ce que l'acteur v soit atteint ou que tous les acteurs accessibles depuis u aient été visités.

La deuxième version de distance, utilise une approche de parcours en profondeur (DFS) pour calculer la distance entre u et v. Cette version emploie une pile pour explorer les voisins de manière récursive jusqu'à ce que l'acteur v soit trouvé ou que tous les acteurs soient visités.

La troisième version de distance, reprend une approche de parcours en largeur (BFS) comme la première version, mais elle est implémentée de manière légèrement différente. Plutôt que d'utiliser une file unique, cette version utilise deux files pour marquer chaque niveau de la hiérarchie du graphe. Cela permet de parcourir le graphe de manière plus efficace en évitant les niveaux inutiles.

Temps pour distance(G, “Jack Lemmon”, “Charles Durning”) sur le jeu de donnée 10 000:
V1: 0.2 secondes
V2: 0.04 secondes mais le résultat est incorrect
V3: 0.0 secondes (le temps est trop proche de zéro pour être affiché)

centralite :

Pour cette fonction, nous avons initialement utilisé un parcours en largeur (BFS) qui, pour traiter les nœuds et les arêtes voisines, "coloriait" le graphe en utilisant des fonctions préalablement créées (traiter, visiter, etc.). Bien que ce programme fonctionnait, sa complexité était beaucoup trop élevée. Nous avons donc réfléchi à une solution plus efficace et moins coûteuse.

Finalement, nous avons décidé d'intégrer une file, un ensemble pour l'optimisation, et un dictionnaire pour stocker et comparer les données facilement. La file permet de gérer efficacement les voisins de chaque nœud et de s'y retrouver aisément en cas de problème.

Le parcours en largeur est la pièce maîtresse de cet algorithme car il permet de parcourir les voisins de chaque nœud de manière progressive sans passer par des cycles. Comparé au parcours en profondeur, le BFS est bien plus adapté pour cette tâche, car il garantit que nous trouvons la distance la plus courte entre le nœud de départ et tous les autres nœuds du graphe. Cette méthode optimise le temps de traitement et réduit significativement la complexité du programme.

Cette version est donc la version final, voici une comparaison des temps de la V1 et la V2 sur le jeu de donnée 10 000:
V1: 2.1 secondes
V2: 0.5 secondes

centre_hollywood :

Cette fonction, qui utilise directement la fonction précédente, peut être très coûteuse si elle n'est pas optimisée. En effet, avec la version initiale (V1) de la fonction centralité, le temps d'exécution était de 40 secondes pour le jeu de données de 100 acteurs. Cela était dû au fait que le graphe devait être réinitialisé à chaque itération de la boucle (pour recalculer la centralité de chaque acteur et les comparer), ce qui devenait problématique.

Après avoir optimisé la fonction centralité, l'exécution de la fonction "centre_hollywood" est devenue beaucoup plus rapide. Les améliorations apportées à centralité, telles que l'utilisation d'une file d'attente, d'un ensemble pour l'optimisation et d'un dictionnaire pour stocker et comparer les données, ont permis de réduire considérablement le temps de traitement. Grâce à ces optimisations, la fonction "centre_hollywood" est désormais plus performante et gère les grands jeux de données de manière plus efficace.

Voici un exemple avec le jeu de donnée 100:
V1: 41 secondes (très long)
V2: 2.9 secondes (bien mieux)

eloignement_max :

Pour cette fonction nous avons réalisé 5 versions. La première version de la fonction eloignement_max parcourt toutes les paires de nœuds dans le graphe G pour calculer la distance maximale entre elles. Cela se fait en calculant la distance entre chaque paire de nœuds à l'aide de la fonction distance.
La deuxième version de eloignement_max, utilise une approche de parcours en largeur (BFS) pour trouver la distance maximale entre toutes les paires de nœuds. Elle parcourt chaque nœud du graphe une fois et utilise un BFS pour trouver la distance maximale à partir de chaque nœud.
La troisième version de eloignement_max, est similaire à la deuxième, mais elle utilise la fonction single_source_shortest_path_length de NetworkX pour trouver la distance maximale entre toutes les paires de nœuds. Cette version est la plus rapide mais nous l'avons retirée car elle utilise une fonction de la bibliothèque networkX.

La quatrième version de eloignement_max, utilise une autre approche en utilisant une fonction auxiliaire bfs_distance_maximale pour effectuer un

parcours en largeur (BFS) à partir de chaque nœud pour trouver le nœud le plus éloigné et sa distance maximale. Ensuite, elle parcourt à nouveau tous les nœuds pour obtenir la distance maximale entre toutes les paires de nœuds

Enfin, la cinquième et dernière version de eloignement_max, qui est notre version finale, utilise la fonctioncentralite pour trouver le nœud le plus éloigné et la distance maximale. Cette version améliore considérablement l'efficacité en évitant les recalculs inutiles et en utilisant une approche optimisée pour trouver les distances maximales. Elle se base sur un parcours en largeur (BFS) pour déterminer les distances, ce qui permet de traiter efficacement les nœuds et leurs voisins. Grâce à ces optimisations, cette version est bien plus performante pour les grands graphes.

Temps pour eloignement_max(G) pour le jeu de données 100:

V1: 212 secondes

V2: 2.4 secondes mais le résultat est faux

V3: 0.001 secondes (mais avec NetworkX)

V4: 50.18 secondes

V5 : 7.9 secondes

Comme je l'ai expliqué au début de la SAE nous avons fait pour les fonctions "distance, centralite, centre_hollywood" et "eloignement_max" des versions optimisées utilisant la bibliothèque networkX, elles sont donc moins coûteuses que les fonctions que nous avons rédigé "à la main".

Conclusion :

Après deux mois de travail intensif sur cette SAE 2.02, nous avons réussi à réaliser les différentes fonctions demandées et à les optimiser au mieux de nos capacités. Nous avons développé des versions évolutives pour chaque fonction, résolvant les problèmes rencontrés au fil du processus et aboutissant à des solutions optimisées grâce à l'utilisation judicieuse de la bibliothèque Networkx.

Nous avons également créé un environnement de test complet pour évaluer nos fonctions, en utilisant des jeux de données plus simples que ceux fournis

avec le sujet pour garantir leur fonctionnement correct. De plus, nous avons développé une interface utilisateur simple sur terminal, permettant à l'utilisateur de choisir les données souhaitées et d'accéder à différentes options telles que le calcul de la distance entre deux acteurs ou la centralité du graphe sélectionné.

En ce qui concerne la répartition des tâches, nous avons travaillé en étroite collaboration, contribuant tous les deux à la création et à l'optimisation des fonctions. Nous avons ensuite réparti les tâches autres que la rédaction des fonctions et de l'optimisation de manière équilibrée, comme illustré dans le tableau présenté.

Pour répondre aux différentes questions posées sur le sujet, nous avons fourni des explications détaillées et des analyses sur les approches utilisées, les complexités algorithmiques et les performances des fonctions. Nous avons également comparé nos résultats avec des solutions disponibles dans la bibliothèque Networkx, mettant en évidence l'efficacité de nos implémentations optimisées.

Enfin, notre évaluation expérimentale a permis de mettre en lumière les différentes versions de nos fonctions, leur évolution et leur efficacité respective en termes de temps d'exécution. Nous avons également souligné l'importance des versions optimisées utilisant Networkx, offrant des performances supérieures tout en conservant la précision des résultats.

En somme, cette SAE a été une expérience enrichissante qui nous a permis d'approfondir nos connaissances en algorithmique, en théorie des graphes et en programmation Python, tout en développant des compétences pratiques précieuses pour résoudre des problèmes complexes.