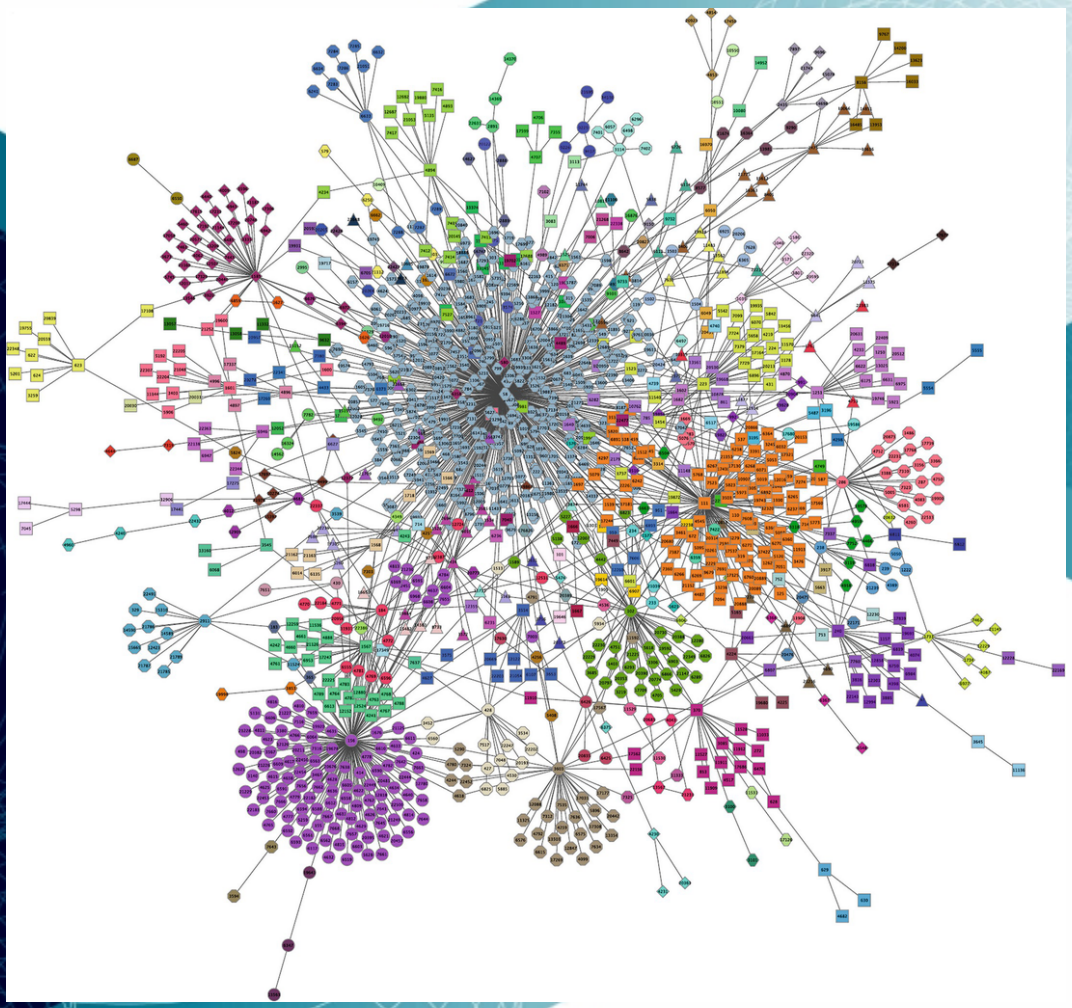
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE



Rapport de Projet Intégrateur
Détection de communautés
UF Projet SDBD

Denis Gouvine-Birrer, Siméon Gaumart, Arnaud Prieu, Lancelot Poulin, Nicolas Chabeau

INSA Toulouse - Mercredi 20 Janvier 2021 - PROMOTION 54 - 5 SDBD

Table des matières

1 Introduction

De nos jours, les individus sont liés entre eux par de nombreux éléments, que se soit à travers les réseaux sociaux, leurs activités, leurs passions, leur travail ou bien d'autres choses. La société et la nature humaine encouragent ces relations entre les individus et leur permet de se relier entre eux par des points communs. Il est alors possible de regrouper ces personnes par groupe, par communautés en analysant les liens qui les relient entre eux. Il est alors légitime de se demander à quelle communauté appartient chaque individu.

Les réseaux sociaux ont pris une part prédominante dans nos vies et dans notre rapport aux autres. Ces interactions ont de plus en plus tendance à augmenter et cela et encore plus favorisé en ces périodes où les interactions sociales directes sont réduites au strict minimum. L'étude de ces réseaux constitue un champ de recherche de plus en plus vaste, avec ses propres objectifs et contraintes. Des technologies ont été développées et continuent d'évoluer dans ce sens. Comme le programme "HelloWorld" serait un premier pas vers la découverte d'un langage de programmation, la principe de détection de communautés est elle aussi un des principaux axes de développement du domaine. Dans ce rapport, nous allons considérer les graphes issus de réseaux sociaux.

Cependant, derrière le terme assez général de "Réseaux sociaux" se cache le principal problème auquel nous sommes confronté, qui est la quantité des données stockées. C'est pour cela que nous avons besoin de certains outils qui nous permettent de traiter quelques milliards de personnes et de relations. Pour ce qui est de la modélisation des différents individus et des relations qui les relient, nous utilisons une structure de données en graphe. Dans ce projet, un graphe est composé de nœuds et d'arcs où chaque nœud représente un individu et où un arc représente une relation particulière entre deux individus, cette relation peut être un lien amical, une relation commerciale, une discussion par message... En somme la relation représentée entre deux individus peut être de nature très variée en fonction des données et de l'information que nous souhaitons représenter. Nous donnons un exemple d'un tel graphe dans la figure 1.

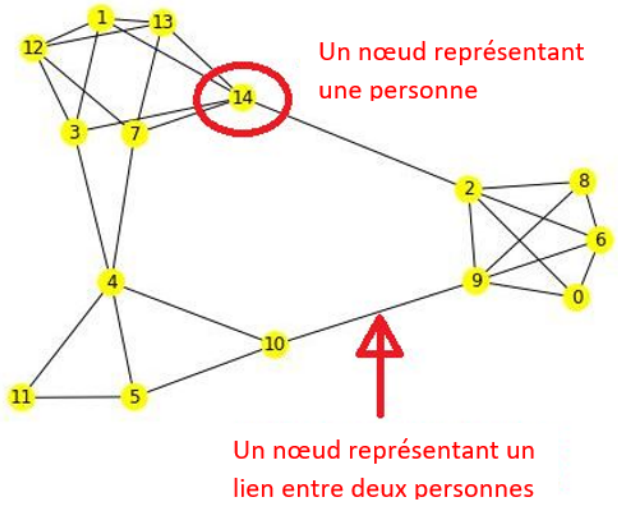


FIGURE 1 – Représentation d'un graphe

Si on considère chaque noeud en comptant le nombre de relations on obtiendra une loi de puissance obtenue dans la figure 2. Cette propriété est vérifiée quelle que soit la taille ou la nature du graphe de relation considéré. Cela dégage une propriété intéressante. Il existe un petit nombre de noeuds qui concentre un grand nombre de relations. Ces noeuds seront appelés "Hubs".

Enfin, il existe au sein de ces graphes des communautés, c'est-à-dire des sous-graphes, qui ont une densité de relation supérieure à un sous graphe pris au hasard et de même taille. Par mesure de simplification du problème, nous allons considérer que chaque noeud des graphes étudiés n'appartiennent qu'à une seule communauté.

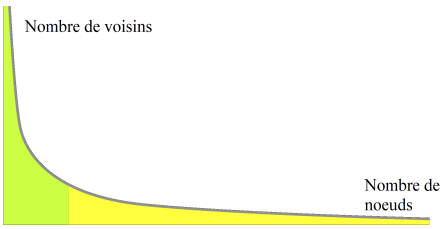


FIGURE 2 – Graphe de loi de puissance

Dans de ce projet, notre principal objectif est de détecter les différentes communautés qui sont présentes dans ces graphes. Cela signifie donc que nous devons déterminer toutes les communautés dans un graphe et que pour chacune d'entre elles nous devons trouver toutes les personnes appartenant à chaque communauté et donc qui la constitue. Dans la figure 3 nous donnons un exemple de résultat d'un algorithme de détection de communautés obtenu à partir du graphe de la figure 1 où chaque couleur permet d'identifier une communauté différente.

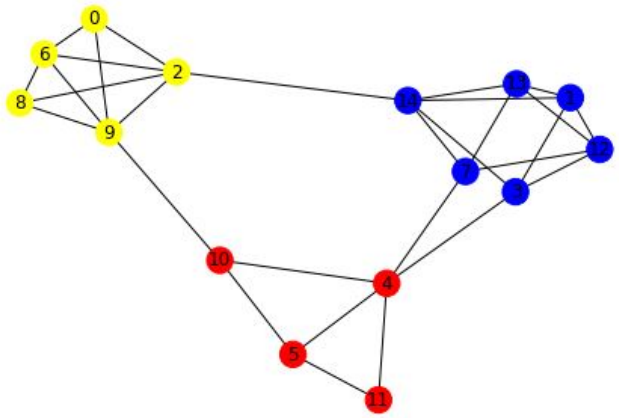


FIGURE 3 – Graphe par communautés

Dans le but de résoudre un tel problème, et notamment sur des jeux de données réelles trouvés sur Internet nettement plus complexe que les données utilisées pour créer le graphe d'exemple et qui contiennent eux quelques milliards de nœuds et des millions de communautés. Nous utilisons des jeux de données et des méthodes d'analyse pour les graphes tels que Neo4j, Apache Giraph et GraphX. Nous utilisons aussi différentes méthodes pour effectuer la détection de communautés comme les méthodes d'apprentissage ou les méthodes d'analyse des graphes.

Le second objectif de notre projet est de comparer les différentes technologies et méthodes que nous avons utilisées sur différents graphes, de taille ou de nature différentes. Nous souhaitons également les classer en terme de vitesse, d'exactitude et d'autres mesures.

Notre dernier objectif est de créer un web service pour notre projet dans le but qu'un utilisateur puisse accéder à toutes les technologies, méthodes et données par l'intermédiaire d'un service web. Afin de le réaliser, les tâches à effectuer sont les suivantes :

— Importer ou créer un graphe
— Visualiser un graphe
— Choisir le méthode de détection de communautés
— Choisir l'architecture utilisée pour la détection des communautés
— Analyser le graphe
— Détecter les communautés dans ce graphe

Lien GitHub pour accéder au code source du projet : `https://github.com/panipanipa/ProjetCommunautaire`

2 Etat de l'art

Dans cette partie, nous allons présenter les algorithmes ainsi que les technologies utilisées dans le secteur. Nous aborderons plus longuement celles que nous avons utilisé pour le projet.

2.1 Technologies

2.1.1 Apache Spark GraphX

Le framework Apache Spark est un ensemble de composants permettant de traiter de larges volumes de données de manière distribuée. Le principal avantage de ce moteur est sa vitesse, beaucoup plus élevée que Hadoop, ainsi que sa simplicité d'usage. Le moteur est codé en Scala, mais on peut développer des applications en Java, R, ou Python.

GraphX est un composant de Spark pour le traitement en parallèle de graphes. Il introduit la classe *Graph* permettant de stocker un graphe depuis un fichier CSV ou texte, et qui a comme propriétés les arrêtes et sommets sous forme d'objet *RDD*. La classe *Graph* contient plusieurs méthodes tel que *subgraph* pour les sous-graphes ou encore *joinVertices*. GraphX comprend aussi une grande collection de générateurs de graphe ainsi que d'algorithmes pour les analyser.

2.1.2 Neo4J

Neo4j est une base de donnée orientée graphe. Ainsi, le problème du stockage n'est pas à penser. Cette base de donnée offre un langage similaire à SQL pour récupérer des noeuds ou relations : Cypher. Cette technologie offre plusieurs librairies intéressantes pour ce projet notamment la librairie *graph data science* qui offre de nombreuses fonctions de détection de communautés ou d'analyse de graphe.

Dans ce projet, nous utilisons Neo4j community edition, version 4.2.0 qui fonctionne avec java 11. Nous utilisons aussi l'interface Neo4j Browser, disponible à l'adresse http ://192.168.37.54 :50003, qui permet d'administrer la base de données.

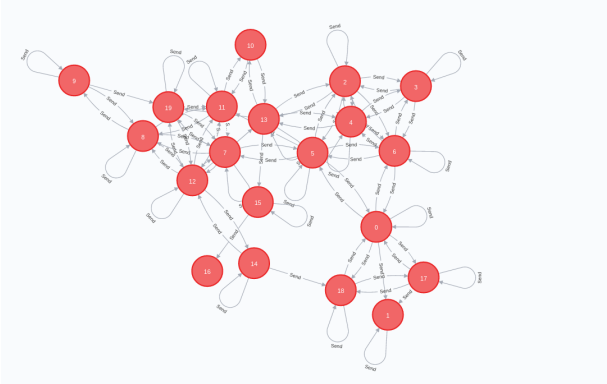


FIGURE 4 – Visualisation des premiers noeuds pour la dataset d'email

2.1.3 Giraph

Apache Giraph est un projet Apache destiné à utiliser l'implémentation MapReduce de Hadoop afin d'appliquer des algorithmes sur de grands graphes. Elle est basée sur l'implémentation google Pregel. Giraph est connu pour avoir été utilisé avec succès par Facebook afin d'analyser leur graphe de relation (environ 10^{12} connexions) à l'aide d'environ 200 machines en quelques minutes. Il a été compliqué de réussir à implémenter un simple job à l'aide de cette librairie.

2.2 Algorithmes

Nous avons appliqué notre projet en étudiant les méthodes servant à détecter les communautés et en établissant des métriques afin de faire des comparaisons entre ces méthodes. Nous avons effectué une distinction entre les méthodes directement appliquées pour trouver les communautés et les méthodes utilisées afin d'estimer la

perspicacité des communautés données. On peut noter que ces algorithmes peuvent parfois être utilisés dans les deux cas.

2.2.1 LabelPropagation

Cet algorithme consiste à considérer le graphe comme un réseau distribué. On va alors boucler sur chaque noeud afin que ceux-ci découvrent leurs voisins et se considèrent comme faisant partie de la communauté la plus présente autour d'eux. Cet algorithme est connu pour avoir de bons résultats, et a l'avantage d'être indépendant de l'implémentation. Cependant, il manque en consistance car l'initialisation et les égalités sont gérées grâce à de l'aléatoire. De plus, son état final n'est pas stable et il faut réussir à détecter les boucles ou l'arrêter de façon arbitraire.

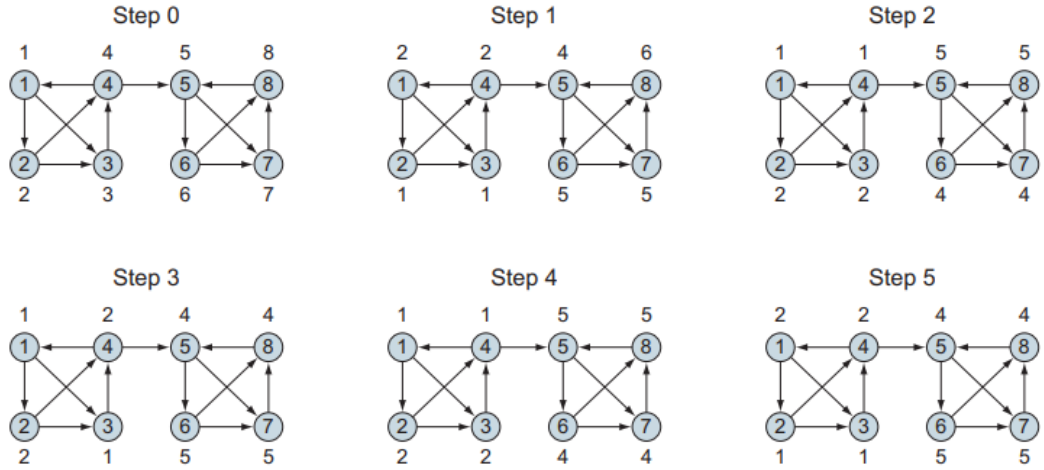


Figure 5.11 The LPA algorithm often doesn't converge. Step 5 is the same as step 3, meaning that steps 3 and 4 keep repeating forever.

FIGURE 5

2.2.2 Louvain et modularité

L'algorithme de Louvain a été pensé afin de maximiser une métrique permettant de mesurer la pertinence d'un découpage en communauté : la modularité. La modularité exprime la densité de liaison intra-communauté, cela comparé à la densité de liaison moyenne en dehors de la communauté. Sa valeur peut varier entre -0.5 et 1. Bien optimisé, sa complexité est $O(n \cdot log_2 n)$.

Bien que nous ne l'ayons pas utilisé, il existe des variantes où l'ordre de passage initial des noeuds est déterminé par des seeds. En choisissant les bonnes seeds (c'est-à-dire des Hubs), on pourra alors obtenir un résultat plus persistant.

2.2.3 TriangleCount

TriangleCount est une métrique qui est distinguée en 2 valeurs : Les coefficients de clusterings locaux et globaux. Le coefficient de clustering local associé à un noeud mesure le nombre de connexions directes entre voisins d'un noeud particulier. Cette valeur peut être globalisé à toute une communauté afin d'obtenir une métrique sur la pertinence d'une communauté. Si on considère trois noeuds a,b et c au sein d'une communauté, avec la relation R étant l'existence d'un lien direct entre deux noeuds, alors le coefficient de clustering globalisé représentera la probabilité P(aRc| $aRcandbRc$)

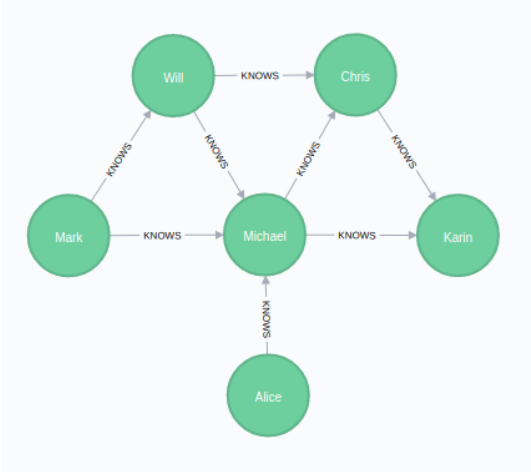


FIGURE 6 – Exemple du coefficient des triangles

Dans cet exemple, celui qui a le plus de triangles connectés est Michael avec 3. Cependant, Mark et Karin ont un meilleur coefficient de clustering car toutes leur connaissances se connaissent entre elles. On peut se rendre compte des caractéristiques de cette technique : plus un noeud aura de connexions, plus une important coefficient de clustering local indiquera la présence d'une communauté resserée.

3 Demarche et Organisation

Nous allons maintenant présenter en détail notre démarche et notre organisation au sein du projet.

3.1 Outils d'organisation

3.1.1 Jira

Jira est un outil de gestion de projet permettant de mettre en oeuvre la méthode AGILE. Il nous a permis de nous répartir les tâches, de fixer à chacun des objectifs à cours terme afin de se rendre compte de l'avancement du projet, lors de *sprints* dont nous avons définis la durée à une semaine. A la fin de ceux-ci, nous avons pu fixer des réunions pour que chacun témoigne de son avancement au groupe et fasse part des difficultés qu'il a rencontré. Cela permet alors de réorganiser les tâches pour le sprint suivant.

3.1.2 OpenStack

Ce projet nécessite d'avoir des capacités de calcul élevées. En effet, nous étudions des grands graphes, et un ordinateur personnel peut mettre des heures voire des journées pour faire fonctionner un algorithmes de détection de communauté. Nous avons donc opté pour l'utilisation de OpenStack, qui est une plateforme où l'on peut utiliser des VM qui vont tourner sur des ordinateurs de l'INSA afin d'allouer les capacités de calcul nécessaires au lancement de nos algorithmes en temps raisonnable. En outre, cela facilite aussi le travail en groupe, car nous avons pu, accéder à ces VM en parallèle et à distance.

3.1.3 Github

Pour la partie codage des algorithmes et gestion des serveurs, nous avons créé un projet Github qui permet de mettre en commun le code de chacun, et aussi de paralléliser le travail en évitant les problèmes de synchronisation posés par le travail en parallèle. Celui-ci est disponible à cette adresse :
https://github.com/panipanipa/ProjetCommunautaire

3.1.4 Jenkins

En liaison avec Github, nous avons créé un serveur Jenkins qui prend en charge l'intégration continue. Cela permet notamment de mettre à jour automatiquement les serveurs lorsque l'on pousse du code afin que cela ne se fasse pas manuellement.

3.1.5 Discord

Discord est un application où il est possible de créer des serveurs permettant la communication écrite et vocale de façon organisée, ainsi que l'envoi de fichiers. Nous avons utilisé cet outil tout au long du projet pour communiquer et s'organiser (la rencontre physique étant très difficile au vu des circonstances liées au coronavirus).

3.2 Organisation temporelle

3.2.1 Répartition des tâches

Au début du projet, les premières séances, nous avons essayé d'installer les différentes technologies, Neo4j, Giraph et GraphX sur des VMs openstack. Denis et Siméon se sont occupés de Neo4j, Arnaud de Giraph et Lancelot et Nicolas de GraphX. Après avoir réussi pour Neo4j et GraphX, à l'issue de quelques séances, et avoir échoué sur Giraph à cause de quelques difficultées, nous nous sommes réparti les tâches de la façon suivante :

- Denis sur l'execution/implémentation des algorithmes sur Neo4j ainsi que du micro-service pour Neo4j et de la gestion de l'équipe.
- Siméon sur la recherche/implémentation/analyse des algorithmes d'apprentissage non-supervisé (clustering) ainsi que la gestion des livrables/soutenances.
- Lancelot sur l'execution/implémentation des algorithmes sur GraphX.
- Arnaud sur la recherche d'algorithme de détection de communautés ainsi que l'intégration continue avec Jenkins.
- Nicolas sur la recherche d'algorithme de détection de communautés.

Noua avons aussi essayé de se rencontrer virtuellement (en vocal) assez souvent afin de faire des points sur les travaux de chacun, l'avancée dans le projet et s'organiser pour la suite. Nous avons aussi chacun participé aux différents rendus qui se sont révélés assez chronophages.

3.2.2 Difficultés rencontrées et adaptation

Durant ce projet, nous avons été confrontés à des difficultés concernant la découverte de certaines technologies (comme Giraph par exemple), concernant l'organisation à cause de la situation de confinement handicapant mentalement et de façon pratique, et de la charge de travail parallèle. Nous avons fait tout de même de notre mieux pour avancer convenablement mais pensons que l'aboutissement aurait pu être mieux dans des conditions plus normales et favorables.

4 Implémentations et Résultats

4.1 Implémentation de Neo4j

4.1.1 L'importation des graphes

L'importation des graphes a toujours été compliqué. En effet, malgré que Neo4j offre la fonctionnalité d'importer des graphes depuis un fichier csv, ces fichiers demandent une syntaxe particulière, ce qui complexifie la requête. Qui plus est, il nous a été impossible d'importer de datasets d'une grande taille. En effet, il semble que le serveur essaye de créer la requête totalement en mémoire pour ensuite commencer à écrire dans la base de donnée. Il est possible avec la commande *Use periodic commit* de faire en sorte que la commande écrive dans la base de donnée après avoir lu n lignes du fichier csv (1000 par défaut). Cette commande n'a jamais fonctionné pour des raisons inconnues. Ce problème semble dépendre de la version de Neo4j utilisé.

4.1.2 Les graphes en mémoire

Pour lancer les différents algorithmes de détection de communauté, nous devons avec Neo4j créer des graphes en mémoire. Un graphe en mémoire est une représentation temporaire du graphe étudié. C'est nécessaire pour lancer plusieurs algorithmes en parallèle sur le graphe ou pour réutiliser un résultat précédent. Les différents mode de fonctionnement nous permet d'obtenir le résultat sous les formes suivantes :

- stream : le server renvoie les résultats directement par ligne, à la manière des requêtes SQL.
- mutate : le server stocke le résultat par noeud dans une nouvelle propriété de celui-ci. Cela permet la réutilisation des résultats. Nous pouvons par la suite récupérer les valeurs des propriétés souhaitées par stream.

— write : ce mode permet de sauvegarder les résultats sur la base de donnée.

Nous utilisons un dernier mode : estimate. Celui-ci estime le coût en mémoire de l'exécution du programme.

Un des premiers problèmes rencontré est le fait que Neo4j n'utilise que des graphes orientés. Nous pouvons configurer la création du graphe en mémoire pour le rendre soit orienté soit non-orienté. Cependant, cela coûte en mémoire et en complexité. Surtout si nous essayons de créer un graphe non-orienté depuis un graphe orienté (cas qui nous avons rencontré pour le dataset d'email et pour lancer certains algorithmes).

	Graphe mémoire orienté	graphe mémoire non-orienté
Graphe Orienté	Pas de problème	On doit fusionner certaines relations entre elles après les avoir doublées
Graphe Non orienté	on double le nombre de relation	on double le nombre de relation

FIGURE 7 – Tableau récapitulatif des coûts pour les graphes en mémoire pour les différents cas

4.1.3 Résultat pour Neo4j

Nous appliquons les algorithmes sur le dataset d'email. Nous considérons les cas où le graphe est orienté et aussi non-orienté. Voici les résultats obtenus (LP signifie Label Propagation) :

	Nombre de communauté	Temps de calculs (ms)	taille Max	taille Min
Louvain sur graphe orienté	118	2726	207	1
Louvain sur graphe non-orienté	27	2892	228	1
LP sur graphe orienté	183	80	739	1
LP sur graphe non-orienté	20	24	986	1

TABLE 1 – Table de résultat pour la technologie Neo4J

La simple distribution des communauté nous permet de constater que Label Propagation ne donne pas de bons résultats : il trouve une communauté énorme . Nous expliquons cela par deux faits :

— Le cas des personnes qui n'ont envoyé qu'un email à elle-même donne des communautés unitaires
— Les noeuds sont trop inter-connecté pour être discriminant

Pour évaluer la pertinence des communautés pour Louvain, nous regardons deux choses :

— Le taux de présence maximale d'un département à l'intérieur d'une même communauté. cela nous permet de voir si la communauté est cohérente entre elles (avec l'hypothèse que les personnes d'un même département ont plus de chance d'appartenir à la même communauté).
— Le nombre de triangle moyen par communauté. Si notre communauté est bien formé, ce nombre devrait être normalement élevé.

Nous faisons cette analyse sur notre application et voici les résultats que nous obtenons :

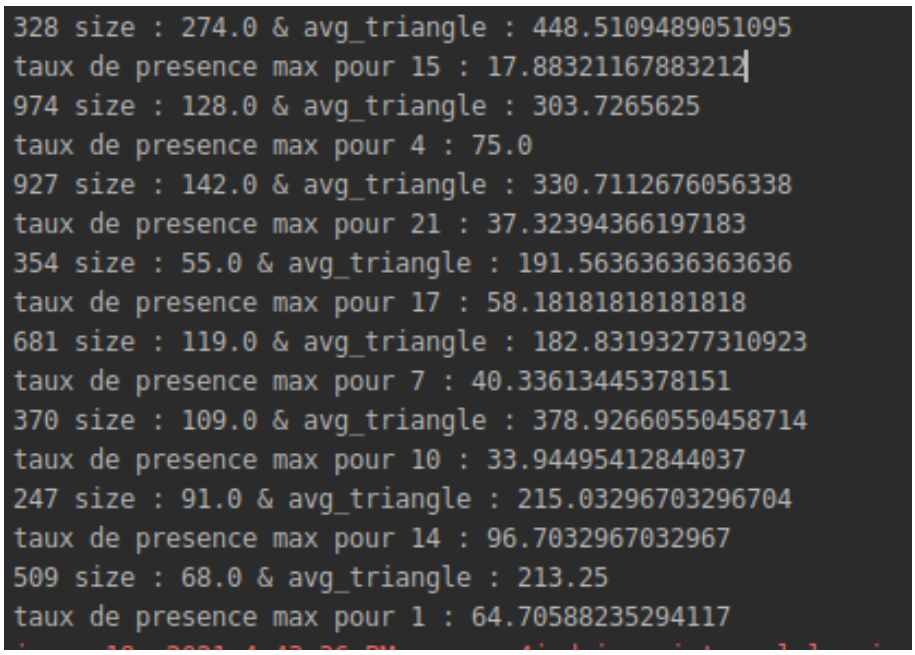


FIGURE 8 – taux de présence max et nombre de triangle moyen

Nous constatons que certaines communautés détecté semble valides. En effet, il y a des communautés assez cohérente (la 274 par exemple avec 96,7% de personne issu du département 14) et il y a des communautés avec plus de 200 triangles pour un nombre total de triangles dans le graphe de 105465.

Pour cette analyse, nous avons dû utiliser la fonction *triangleCount* qui c'est exécuté en 16ms.

4.2 Implémentation de GraphX

4.2.1 Instanciation d'un objet Graph

Lors du lancement du serveur Apache Spark, on obtient une variable SparkContext qui est le point d'entrée de Spark, et qui nous donne les principales méthodes pour créer un Graph, charger un fichier ou encore paralléliser les calculs. Nous utilisons le langage par défaut Scala qui s'approche du langage Java. Un fois l'importation du fichier exécuté, nous pouvons créer un objet de graphe Graph en précisant le séparateur entre les nodes qui varie selon le format du fichier : un espace, une virgule ou une tabulation.

4.2.2 Le niveau de stockage

Spark nous permet de gérer différents paramètres pour les données chargées. On peut choisir le nombre d'octets maximum ainsi que la persistance dans la classe StorageLevel :

- MEMORY ONLY : Les données sont stockées en cache de manière désérialisé. C'est le paramètre par défaut et celui qui permet les meilleures performances, mais peut causer des "out of memory" si le RDD est trop grand.
- MEMORY AND DISK : Les données sont stockées en cache et en dur sur le disque lorsque le cache n'est pas disponible.
- DISK ONLY : Les données sont stockées en dur sur le disque. Beaucoup moins performant, il ne faut pas utiliser cette option si l'on utilise souvent les données car les entrées sorties du disque peuvent être surchargées.

Il est aussi préférable de stocker les RDD de Spark dans une base de données NoSQL tel que MongoDB ou Cassandra lorsque Spark est éteint. Cela permet de ne pas recharger les datasets depuis un fichier ce qui peut prendre beaucoup de temps pour les grands graphes.

4.2.3 Label Propagation

GraphX implémente de base plusieurs méthodes de détection de communauté, tel que Label Propagation et Strongly Connected Components. Les résultats de cette dernière n'est pas probant car les noeuds des graphes

utilisés sont trop reliés entre eux.

4.2.4 L'algorithme de Girvan et Newman

Nous avons implémenté dans GraphX l'algorithme de Newman qui utilise la edge-betweenness pour déduire des groupes. C'est une mesure de centralité, c'est à dire le nombre de plus courts chemins de chaque noeud. Les arrêtes ayant la plus grande centralité sont plus susceptible d'être des ponts entre les communautés et sont donc supprimées.

Les résultats sont très satisfaisant dans des petits graphes mais la complexité de l'algorithme le rend inutilisable dans les grands graphes. Les expériences ont été menées avec une VM Ubuntu sur une machine Windows.

Nombre d'arêtes	Temps d'exécution
100	1 seconde
1000	45 secondes
5000	2 heures 20 minutes
25571	6 jours 5 heures

Table 2 – Temps d'exécution de Newman

On observe un temps d'exécution d'ordre exponentielle et ils faudraient plusieurs calculateurs pour obtenir un bon résultat dans des grands graphes en un temps raisonnable.

4.2.5 Méthode de Louvain

La méthode de Louvain permet une résolution à multiples niveaux où l'on va maximiser la modularité Q locale du groupe et non pas celle du graphe entier. Pour chaque noeud, on cherche là où il augmente le plus la modularité du groupe. Chaque itération est un niveau où l'on obtient une partition des groupes. On arrête lorsqu'il n'y a plus de gain positif issus de la fusion des communautés.

L'algorithme est configurable : on peut changer le parallélisme, le nombre de sommet qui doivent être déplacé à chaque étape pour progresser, ou encore le nombre de fois ou on tente d'obtenir une meilleure solution avant de stopper l'algorithme. On observe un temps d'exécution bien plus raisonnable que celui de Newman et même dans des très grands graphes. Les expériences ont été menées avec une VM Ubuntu sur une machine Windows.

Graphe	Nombre d'arêtes	Nombre de noeuds	Temps d'exécution	Niveaux
Email	25571	1005	14.16 secondes	2
Amazon	925872	334863	882.07 secondes	5

Table 3 – Temps d'exécution de Louvain

Si un graphe a un score de modularité Q supérieur à 0.3 alors on peut dire qu'il a une structure de communautés significative. Les résultats obtenus sont donc pertinents et on retrouve des communautés réelles.

Graphe	Nombre de communautés	Q-value
Email	44	0.35
Amazon	185	0.81

Table 4 – Résultat de Louvain

Pour le graphe des échanges de mails, nous avons des communautés qui regroupent plus de 50% de personnes d'un même département et on obtient un nombre de 44 communautés pour 42 départements.

Le dataset d'Amazon basé sur "Customers Who Bought This Item Also Bought" a une très bonne modularité Q et nous observons des communautés bien distinctes. On remarque que si on s'était arrêté au niveau 4 de l'algorithme, on avait encore 782 communautés, qui se sont donc fusionnées en 185 communautés au niveau 5.

4.3 Neo4j VS GraphX

La mise en place de ces technologies n'est pas très compliqué. Un premier point qui diffère entre les deux approches est la diversité des algorithmes de détection de communauté entre Neo4j et GraphX. La librairie *graph data science* de Neo4j est assez complète pour l'analyse et va bientôt se voir enrichir (il y a beaucoup de nouvelles fonctions en alpha et bêta). GraphX implémente déjà plusieurs méthodes de détection de communauté et il y a de nombreux projets projets open-source sur Github qui enrichissent ses fonctionnalités mais c'est une gestion supplémentaire.

Deuxième point, Neo4j est une base de donnée ce qui implique une certaine administration de celle-ci. Ce travail d'administration est présent dans GraphX si nous le couplons avec une base de donnée mais il n'est pas intrinsèque à la technologie.

Par rapport à la rapidité, un bref calcul montre que pour le dataset d'email, Neo4j a été six fois plus rapide que GraphX. On ne peut pas être certain que sur un plus grand graphe, cette tendance se vérifie mais on peut penser que Neo4j serait plus rapide. Cette hypothèse ne prends pas en compte le temps de création du graphe et du graphe en mémoire sur Neo4j qui s'ajoute normalement au temps de d'exécution de l'algorithme.

4.4 Méthodes d'apprentissage non-supervisées

Dans cette partie, nous avons essayé d'utiliser des algorithmes de clustering (apprentissage non-supervisé) pour effectuer la détection de communauté. Nous avons essayé 4 méthodes qui sont : Kmeans, Clustering Agglomératif, DBSCAN et HDBSCAN.

Afin de pouvoir essayer ces algorithmes, nous avons du implémenter des méthodes en python permettant de transformer les jeu de données .csv en graphes networkx (librairie de graphes sur python), afin de pouvoir importer des vrais jeux de données, permettant d'afficher un graphes, afin d'avoir un visuel du résultat, et permettant de créer un graphe de communauté aléatoire où 2 noeuds d'une même communauté ont 0.8 chance d'être liés, et 2 noeuds de communautés différentes ont 0.05 (paramètres changeables).

Pour les méthodes de clustering, nous avons utilisé la librairie python scikit-learn. Ces méthodes nécessitant une matrice en entrée, nous avons décidé de transformer les graphes en matrices de la façon suivante : chaque ligne et chaque colonne de la matrice correspond à un noeud et la valeure ligne i, colonne j, est égale à 1 si les noeuds i et j sont liés, et 0 sinon.

Le notebook python contenant les implémentations et les résultats est disponible dans le dossier Clustering du Github.

4.4.1 Kmeans

Tout d'abord, nous avons essayé l'algorithme de clustering kmeans. Nous avons d'abord essayé cet algorithmes sur un petit graphes composé de quelques noeuds afin d'avoir un résultat bien visible. Nous avons pu constater que l'algorithme détecte parfaitement toutes les communautés (en donnant directement le bon nombre de communauté en paramètre), comme sur la figure ci-dessous.

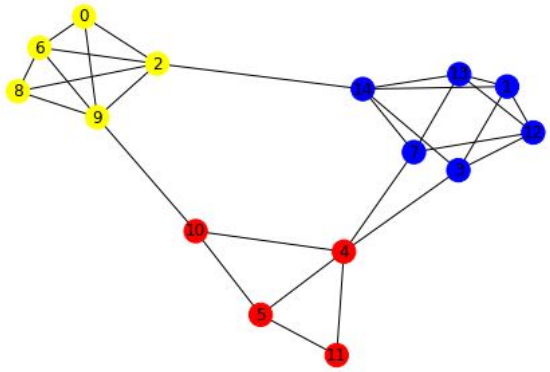


FIGURE 9 – résultat de Kmeans sur un petit graphe

Nous avons ensuite essayé Kmeans sur un graphe aléatoire plus grand, comportant 1000 noeuds et 5 communautés, en utilisant la fonction implémentée avec les paramètres par défaut. On remarque que, encore une fois, la méthode détecte parfaitement les communautés sans erreur, comme sur la figure ci-dessous.

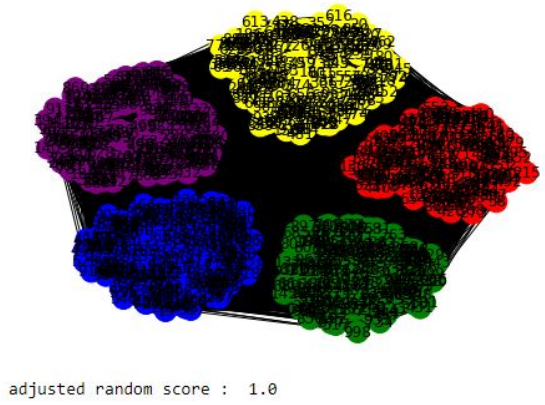


FIGURE 10 – résultat de Kmeans sur un plus grand graphe

Finalement, nous avons essayé cet algorithme sur un vrai graphe, le graphe des emails comportant 42 communautés. Nous avons pu remarquer, que cet algorithme ne fonctionne pas très bien cette fois (en utilisant une telle matrice), et ne détecte pas correctement les communautés, comme le montre la figure ci-dessous.

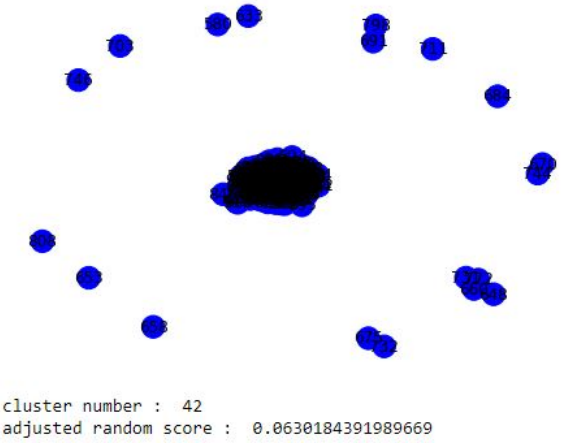


FIGURE 11 – résultat de Kmeans sur un vrai graphe

Nous pouvons remarquer ici que certains noeuds n'ont aucun voisins et sont quand même considérés comme étant dans la même communauté. En effet, ces noeuds ont la même valeur pour chaque colonne dans la matrice : 0. Ils sont donc au même endroit dans le repère de dimension égale au nombre de noeuds, et la distance euclidienne entre eux est nulle. Kmeans va donc forcément les regrouper dans le même cluster. Pour palier a ça, nous avons implémenté 2 méthodes :

- une méthode changeant les 0 des noeuds n'ayant pas de voisins en une valeur comprise entre 2 et 255 (car les valeurs sont codées sur 8bits dans la matrice) de façon aléatoire, et homogènement répartie sur l'espace.
- une méthode créant un arc virtuel aléatoire pour chaque noeud sans voisin.

La première méthode fonctionne à séparer les noeuds sans voisins dans différents clusters, là où la seconde méthode ne fonctionne pas. De plus pour les deux méthodes, la score (comparaison des clusters obtenus avec les clusters réels en utilisant le métric adjusted_rand_score) n'est pas plus élevé finalement, pour la même raison du dysfonctionnement de la deuxième méthode : dans ce jeu de données, beaucoup de noeuds ont très peu de voisins. De ce fait, la distance euclidienne entre eux va être très faible et ils vont être regroupés dans des mêmes clusters. Kmeans fonctionne cependant très bien sur le graphes générés aléatoirement par la méthode avec les

paramètres par défaut, car la densité d'arcs sur ce graphe est suffisamment grande et les noeuds ayant très peu voire aucun voisin sont inexistant.

4.4.2 Clustering Agglomératif

Dans cette partie, nous avons essayé l'algorithme de clustering agglomératif. Nous l'avons d'abord essayé sur le même petit graphe que précédemment, puis sur un plus grand graphe généré aléatoirement, et le résultat reste très satisfaisant peu importe la méthode de linkage utilisée en paramètre de l'algorithme.

Nous avons ensuite essayé cet algorithme, avec les différentes méthodes de linkage, sur le graphe des emails, dont les résultats des scores sont affichés sur la figure ci-dessous.

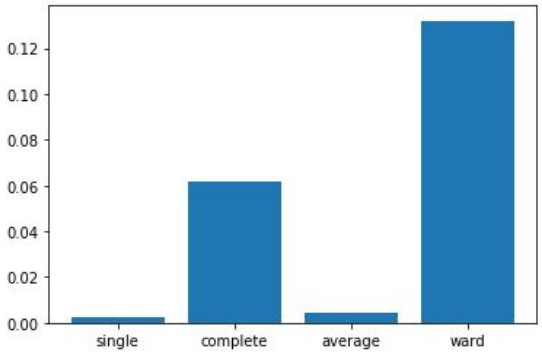


FIGURE 12 – scores de Agglomerative Clustering sur un vrai graphe

On remarque que, comme avec Kmeans, les résultats sont globalement mauvais. On remarque cependant la méthode de linkage qui fonctionne le mieux est ward, là où single et average renvoient un score quasi nul. La raison de ces scores très bas est la même que celle évoquée pour Kmeans. La méthode de linkage ward renvoie un score plus élevé car elle permet de fusionner les clusters entre selon leur variance, là où les autres méthodes utilisent directement la distance euclidienne pour la fusion.

4.4.3 DBSCAN

Dans cette partie nous avons essayé l'algorithme DBSCAN. Comme dans les parties précédentes, nous l'avons essayé sur les trois graphes utilisés précédemment : un petit graphe de quelques noeuds, un plus grand graphes aléatoire de 1000 noeuds, et le graphe des emails. Pour chaque graphe, nous avons itéré sur les paramètres principaux de DBSCAN, eps et min_samples, afin de trouver les meilleurs pour chaque graphes (pour maximiser le score). Nous avons pu remarquer que pour tous les graphes, le min_samples idéal est de 1 et que le eps idéal varie. Nous avons pu remarquer que, pour le graphe aléatoire où la densité d'arc est suffisamment élevée et constante suivant le nombre de noeud, l'eps idéal augmente avec le nombre de noeud dans le graphe. eps est le rayon du cercle où l'algorithme va essayer de trouver un autre noeud, et il utilise la distance euclidienne pour ensuite vérifier si un noeud est dans le cercle (pour min_sample = 1). Or en utilisant cette méthode de transformation du graphe en matrice, la distance euclidienne entre 2 noeuds est proportionnelle au nombre de voisins que les deux noeuds n'ont pas en commun mais qui sont quand même liés à un des deux noeuds. Ainsi plus le graphe est grand, plus cette distance augmente en moyenne.

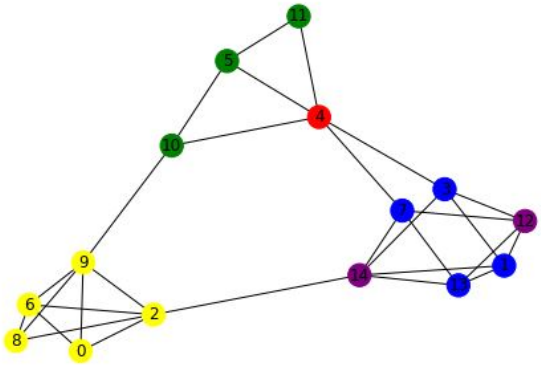


FIGURE 13 – résultat de DBSCAN sur le petit graphe

Nous avons pu remarquer que DBSCAN ne fonctionne pas parfaitement sur le petit graphe (figure ci-dessus) mais renvoie un score correct (vers 70%), là où il reste très efficace sur le graphe aléatoire dense en arc et homogène, et fonctionne très mal sur le graphe des emails avec un score encore plus bas que les algorithmes précédents. En effet, DBSCAN est très sensible aux variations de densité d'arc dans les clusters et à vite fait de fusionner plusieurs clusters, où de séparer tous les noeuds d'un cluster (dans des clusters différents) si la densité varie trop. Avec en plus, toujours le même problème des noeuds avec peu de voisins étant trop proche en distance euclidienne.

4.4.4 HDBSCAN

Le dernier algorithme de clustering que nous avons essayé est HDBSCAN. Encore une fois, nous avons itéré sur le paramètre de l'algorithme, min_cluster_size, afin de trouver le meilleur pour chaque graphe. Nous avons pu constater que les résultats restent très similaire à DBSCAN, où l'algorithme fonctionne un petit peu mieux que DBSCAN sur le petit graphe (comme le montre la figure ci-dessous), parfaitement sur le graphe aléatoire à densité forte en arc, et un peu moins bien que DBSCAN sur le graphe des emails pour encore une fois les mêmes raisons.

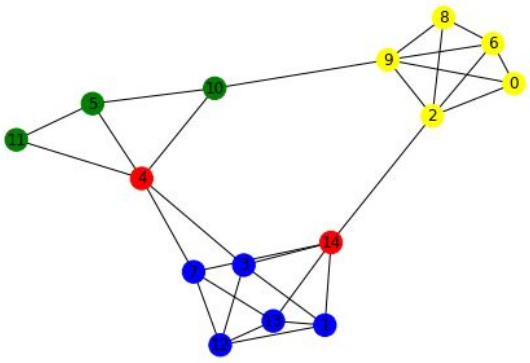


FIGURE 14 – résultat de HDBSCAN sur le petit graphe

4.4.5 Comparaison

Pour finir, nous avons comparé les différents algorithmes de clustering en terme de score et de temps d'exécution sur un graphe généré aléatoirement contenant 1000 noeuds, puis sur le graphe des emails, en choisissant les meilleurs paramètres trouvés dans les parties d'avant pour les différents algorithmes. Les résultat sont affichés dans les figures.

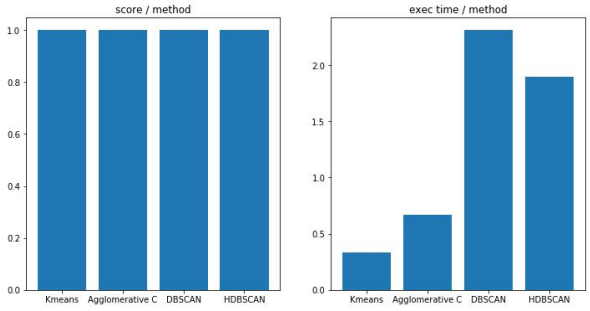


FIGURE 15 – résultats des comparaisons pour le graphe aléatoire

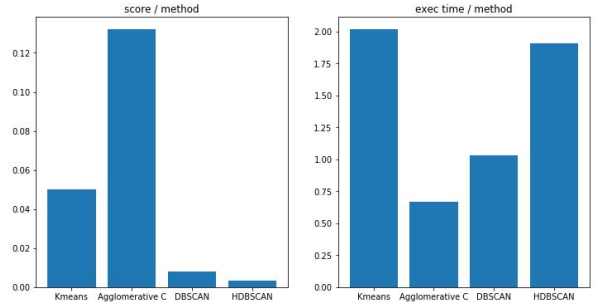


FIGURE 16 – résultats des comparaisons pour le graphe des emails

On remarque que, pour le graphe aléatoire, tous les algorithmes de clustering renvoient un score parfait sur ce graphe, mais que la méthode kmeans est plus rapide. Ce résultat est cependant différent pour le graphe des emails. L'algorithme renvoyant de loin le meilleur score est le clustering agglomératif utilisant la méthode de linkage ward. En effet tous les autres algorithmes utilisés font appel directement à la distance euclidienne pour fonctionner, ce qui ne fonctionne pas avec ce graphe à cause de la trop faible densité d'arcs. De plus, la méthode de clustering agglomératif est plus rapide sur ce graphe que les autres qui convergent plus lentement.

Nous avons donc vu que les méthodes de clustering, en utilisant cette méthode de transformation du graphe en matrice, fonctionnent particulièrement bien sur les graphes ayant de noeuds avec une forte densité de voisins. Cependant, elles fonctionnent très mal pour les graphes ayant des noeuds avec très peu de voisins où pas du tout, en regroupant ceux-ci dans un même cluster. Une perspective d'amélioration serait donc d'utiliser autre chose que la distance euclidienne dans les algorithmes de clustering. On pourrait par exemple implémenter notre propre distance où le nombre de 0 communs ne rapprocheraient pas deux noeuds entre eux, où seulement le nombre de voisins communs ferait réduire le distance. Où comme par exemple la distance en nombre de voisins séparant deux noeuds. De plus, augmenter significativement le nombre de noeud dans le graphe rendrait cette méthode impossible. En effet, la matrice générée est carrée, avec un nombre de ligne et de colonnes égale au nombre de noeuds. Cette matrice serait donc beaucoup trop volumineuse pour des grand graphes : si on prenait un graphe de 1 000 000 de noeuds, il faudrait une matrice de 1 000 000 000 000 d'éléments de 8 bits, ce qui ferait une taille de donnée d'environ 1TO, ce qui n'est pas envisageable.

5 Conclusion

Pour reprendre rapidement le but de ce projet, nous avons essayé de comparer différentes architectures que ce soit avec GraphX sur SPARQL ou en utilisant Neo4j. Nous avons comparé ces 2 architectures pour déterminer quels sont leurs avantages et leurs inconvénients notamment en terme de stockage des données. En plus de comparer les architectures qui nous servent de support, il est tout aussi important de comparer les différents algorithmes de détection de communautés. Nous avons comparé les deux algorithmes suivants, l'algorithme de Louvain et l'algorithme de label propagation en terme de temps d'exécution ainsi que de qualité des communautés détectée. Après avoir effectué nos tests, nous avons trouvé que l'algorithme Label Propagation est plus rapide que la l'algorithme de Louvain mais que le résultat est beaucoup moins précis. Dans l'ensemble, l'architecture Neo4j nous a donné des meilleurs résultats en terme de temps d'exécution que l'architecture GraphX. Neo4j plus efficace pour combiner les graphes Nous avons aussi essayé d'implémenter une solution permettant d'utiliser des algorithmes d'apprentissage non-supervisé et avons notifié de bons résultats sur certains types de graphes, mais de mauvais sur d'autres et certains problèmes ayant des perspectives d'amélioration.

Bien que nous ayons réussi une première approche assez poussée et complexe pour certaines architectures que ce soit sur Neo4j et GraphX, il reste des perspectives d'amélioration afin de poursuivre les comparaisons.

En effet, nous n'avons pas réussi à utiliser l'architecture Giraph et donc nous nous sommes focalisés sur Neo4j et GraphX, cela nous aurait permis d'obtenir un troisième point de vue et ainsi d'avoir des comparaisons plus complètes.

Un autre axe d'amélioration est l'implémentation et l'utilisation d'un autre algorithme de détection de communautés qui est l'algorithme de Girvan-Newman, qui comme l'aurait été Giraph pour les architecture, aurait permis de considérer un autre point de vue et une autre comparaison. Cet algorithme consiste à supprimer certains arc du graphe initial qui sont susceptible d'être des "ponts" entre différentes communautés. En supprimant ces arcs petit à petit, cela permet d'isoler les différentes communautés au fur et à mesure. Une utilisation de cet algorithme sur GraphX, nous a permis de nous rendre compte que les temps d'exécution de cet algorithme augmente très fortement avec le nombre d'arc, bien que les résultat soient très encourageant. Par exemple, pour 25000 arcs, le temps d'exécution est de plus de 6 jours. Nous en avons conclu que nous devions écarter l'algorithme de Givan-Newman de notre étude.

De plus nous pourrions utiliser les métriques TriangleCount et local clustering coefficient dans le but de créer des nouveaux algorithmes de clustering agglomératif.

Une autre perspective d'amélioration aurait aussi été d'intégrer les algorithmes de clustering aux architectures Neo4j et GraphX.

INSA Toulouse
135, avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées
Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE